EFFICIENT AND REFINABLE ATTACK INVESTIGATION

A Dissertation
Presented to
The Academic Faculty

By

Yang Ji

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2019

EFFICIENT AND REFINABLE ATTACK INVESTIGATION

Approved by:

Dr. Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. David Devecsery
School of Computer Science
Georgia Institute of Technology

Dr. Dongyan Xu
School of Computer Science
Purdue University

Dr. Angelos Keromytis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: August 23, 2019

UNIX is basically a simple operating system,

but you have to be a genius to understand the simplicity.

Dennis Ritchie

To my wife Ying and our parents,

who encouraged and supported me throughout this Ph.D. journey.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my thesis committee chair and advisor Dr. Wenke Lee, who has consistently supported and encouraged me on this research. Particularly, without his foresight in cybersecurity or lasting guidance, this dissertation would not have been possible. I would also like to thank Dr. David Devecsery for advising my research with expertise in systems and superb programming skills.

I would like to thank the rest of my committee members, Dr. Alessandro Orso, Dr. Dongyan Xu and Dr. Angelos Keromytis, who have helped me and given me important feedback for improving this thesis.

I owe a great debt to my mentor, Dr. Sangho Lee who gave me great inspiration and significant help.

I would like to thank the fundings from the US Department of Defense, and the School of Computer Science of Georgia Tech for me to complete this research.

I would also like to thank Dr. Taesoo Kim for inspiring the research in record replay systems, and the lab colleagues who have worked with me and contributed on this research including Joey Allen, Mattia Fazzini, Evan Downing, Wilson Martin, Carter Yagemann and Simon Chung. Without their help and input, this research would not be complete.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

As modern attacks become more stealthy and persistent, detecting or preventing them at their early stages becomes virtually impossible. Instead, an attack investigation or provenance system aims to continuously monitor and log interesting system events with minimal overhead. Later, if the system observes any anomalous behavior, it analyzes the log to identify who initiated the attack and which resources were affected by the attack and then assess and recover from any damage incurred. However, because of a fundamental tradeoff between log granularity and system performance, existing systems typically record system-call events without detailed program-level activities (e.g., memory operation) required for accurately reconstructing attack causality or demand that every monitored program be instrumented to provide program-level information.

In this thesis, I will present my research focusing on addressing this issue. First, I will present a **R**efinable **A**ttack **IN**vestigation system (RAIN) based on a record-replay technology that records system-call events during runtime and performs instruction-level dynamic information flow tracking (DIFT) during on-demand process replay. Instead of replaying every process with DIFT, RAIN conducts system-call-level reachability analysis to filter out unrelated processes and to minimize the number of processes to be replayed, making inter-process DIFT feasible. Second, I will present a data flow tagging and tracking mechanism, called RTAG, which further enables practical cross-host attack investigations. RTAG allows lazy synchronization between independent and parallel DIFT instances of different hosts, and applies optimal tag map to minimize memory consumption. Evaluation results show RTAG is able to recover true data flows of realistic cross-host attack scenarios with low time and memory cost. Furthermore, we deployed RAIN and RTAG in the red team adversarial engagements funded by the DARPA Transparent Computing program. The data generated by our system effectively reconstructed the causality of the attacks with high accuracy, even in the presence of knowledgeable attackers.

CHAPTER 1
INTRODUCTION

Cyber attacks have grown to be a pressing issue because a great deal of user data shared and stored in the digitized services are being stolen, tampered and misused, particularly at large vendors and internet service providers. Numerous data breach incidents have affected millions of people, causing their identities to be stolen, and private information such as social security to be exposed [1, 2, 3, 4]. According to the public Gemalto [5] report, on average 1.8 billion records are stolen every year since 2013. The burden on vendors, government and the technology industry to secure the data is higher than ever. For example, in 2018, multiple consumer groups in Europe accused Google and Facebook of not safely protecting user data and requested compensation [6]. Once a data breach happens, a postmortem forensic analysis needs to find out which data have been really stolen, who is impacted, and why the incident happened. Unfortunately, forensic analysis is often complicated by stealthy malware behaviors such as blending with benign activity [7] and staying in-memory only [8]. As a result, users are now torn between the difficulty to defend their systems, and the strong demand for safer systems. With the increasing importance of mitigating these disastrous attacks, researchers explore accurate system monitoring tools while maintaining the system performance without incurring high overhead.

In pursuit of an effective forensic analysis to understand what and why the incident happened and its impacts, many have explored different forensic investigation systems. However, it is challenging to achieve both accuracy in the analysis and efficiency in the system monitoring. Some system monitoring tools efficiently capture course-grained events, like system-call events for analysis, but have no access to the program's fine-grained behavior such as memory operations (Section §2.1 presents the related work of this kind). On the other hand, dynamic program analysis tools provide fine-grained data such as user behavior

at instruction level, but incur an prohibitively high overhead to the program (Section §2.2 presents the related work of this kind). This thesis shows we can achieve the best of these two worlds: an efficient system with low runtime overhead, but with the ability to monitor the program behavior as low as the instruction level.

We observe that the coarse-grained information is usually an abstraction of the fine-grained data, which we can use to guide the fine-grained analysis. For example, by tracking system call trace, one can know when a program starts and ends execution, what and when it performs file IO or network IO operations, etc. Such information provides a valuable clue for what the fine-grained analysis should focus on. For example, when verifying a potential data leakage in a data exfiltration incident, the fine-grained analysis only needs to perform dynamic taint analysis on the part of execution between reading a sensitive user file, and sending data to a remote host, instead of the entire long execution.

We propose using coarse-grained analysis to identify the part of the execution having causal relation to the attack, then perform the fine-grained analysis on those only. The fine-grained analysis is delayed from the program execution runtime without incurring high overhead of instrumentation. Upon a request for analyzing an attack, we perform a search and pruning procedure to narrow down and pinpoint the executions related to the attack, which we call "*refinement*", then we focus on replaying and performing the fine-grained analysis only on these executions at the memory level with single-instruction granularity.

We present a framework called *refinable* attack investigation that brings the benefits of the two worlds: high efficiency in program runtime, and the ability to observe program behavior at fine-grained granularity, even in today's complex multi-host systems. The following thesis statement summarizes the contributions of this document:

Refinement enables the forensic analysis to have fine-grained analysis at the performance cost of coarse-grained analysis, even in today's complex multi-host environments; furthermore, the system is robust and reliable, even when being assaulted by professional hackers.

1.1 Decoupled analysis

In order to maintain an efficient program runtime, we perform the refinement-guided fine-grained analysis at a *delayed* time. We adopt the decoupled analysis to remove the analysis from the program runtime, and delay the analysis until offline time. As a good candidate, record and replay technique such as Arnold [9] achieves this decoupled analysis by efficiently recording the non-deterministic inputs to the program such as file input and data synchronization (less than 10% overhead), then replaying the execution by enforcing the same inputs. Particularly useful for forensic analysis, record and replay can faithfully replicate the same control flow and memory states during replay so the analysis can observe fine-grained user behavior such as memory operations at instruction level. These systems such as Arnold [9] focus on recording individual programs. For the complete coverage for monitoring, *every* program executed in the system is supposed to be logged or recorded. This causes the overhead in analyzing all the program executions to be surged. As a result, the forensic analysis needs to search for and locate the executions related to the attack from high volume of system-wide logs and recorded data. For example, over 20 million system events are generated per day on average on a typical desktop according to Sysdig [10]. Replaying and analyzing all these behavior takes a long time and thus is still impractical. To tackle this challenge, we need *refinement* along with the decoupled analysis that largely reduces the search space and workload of analysis, making the decoupled analysis practical for accurate system-wide monitoring and attack analysis.

1.2 Refinable Attack Investigation

Refinement allows us to make strong claims about past system behaviors, without the high overheads of instrumentation at program runtime. During one analysis, the refinement identifies the executions having causal relationship with the attack and directs the replay to focus on those executions. Following this direction, we first present RAIN, a practical

Refinable **A**ttack **IN**vestigation system, that *selectively* provides an memory-level detailed log while minimizing runtime overhead. RAIN satisfies these conflicting goals using a *system-call-level record-and-replay technology* and *on-demand dynamic information flow tracking (DIFT)*. RAIN continuously monitors and logs system-call events and additional data for later replay while constructing a logical provenance graph. When it detects any anomalous event in the graph, it performs replay-based DIFT from the event to prune out any unwanted dependency. Performing DIFT for every process in the graph, however, is unfeasible because the overhead of DIFT is too high (usually around 10×–20× and at best, 2.7×, using decoupling techniques [11, 12]). Instead, RAIN performs system call-level *reachability analysis* to extract a subgraph tightly related to the anomaly and then conducts DIFT only for processes belonging to the subgraph. The runtime overhead of RAIN is less than 5% on the SPEC CPU2006 benchmark, while the offline analysis is shown to be able to resolve the data dependency explosion problem with dynamic taint analysis and the root cause analysis on memory corruption based attacks. Moreover, we show the effectiveness and performance of RAIN in a red team exercise performed by DARPA.

1.3 Cross-Host Investigation

As the majority of today's systems are networked, attackers have made lateral movement to infiltrate the victim's network and started to steal sensitive data, not from a single host, but with multiple hops of hosts. We extend to consider the case in the networked systems and how we resolve the particular challenges of data dependencies across different hosts. For example, a security team demoed an attack leveraging the `git` protocols [13] to exfiltrate user data from inside an enterprise crossing three hosts, the developer's desktop, the git server, and the attacker's controlled domain. Due to the stateful data communication between hosts, running the refinable system separately on each host cannot effectively recover the linked data causality across multiple hosts. Instead of orchestrating the replays of executions on different hosts, we propose a new mechanism called RTAG, to decouple the dependencies of

data flows from the execution, and make the replays independent with each other, thus they can be performed in parallel. We show this technique reduces the complexity and time cost of replaying for cross-host scenarios by over 60%. In addition, by applying optimal sizes to the tagging of analyzed data, we show lowered memory cost by up to 90% when performing dynamic taint analysis on multiple recorded executions.

1.4 Adversarial Engagements

Security solutions are known to suffer from a "cat and mouse" game, where attackers are constantly racing with defenders. To show that our approach creates a sound, reliable, and real defense against these attacks, we opened our system to assault by a professional red-team group. Our systems managed to survive the attacks without data loss and our generated data reconstructed the attack behavior with high accuracy. We deployed RAIN and RTAG in the red team adversarial engagements funded by the DARPA Transparent Computing program. The results show that our system does not suffer from this cat and mouse game, even in the presence of knowledgeable attackers.

The goal of the red team is to evade the detection and analysis by leaving no trace: they loaded themselves reflectively into memory to avoid executing from disk; system calls were leveraged to avoid commonly monitored APIs such as file `open()` and `write()`; modules were loaded in-memory for network recon, screen capture, audio capture, video capture, and keylogging; and privilege escalation and process injection are introduced to pivot to other processes. The RAIN and RTAG systems were customized and deployed in such adversarial environments to generate data of system events for a set of teams to analyze and detect attacks. Besides the system-call-level events, we also provide a query interface for the analysis teams to query for the fine-grained causality information based on the results of replay and dynamic taint analysis. We have successfully achieved these two goals. The results of the engagements show the analysis teams are able to detect every attack activity using our data. In addition, by making fine-grained queries to our replay and dynamic taint

system, the analysis teams are able to improve the detection accuracy by removing false positive cases.

1.5 Exploration on New Record Replay System

Though the record replay technique achieves low-overhead recording, the fine-grained forensic analysis for auditing the replayed execution is unnecessarily slow, because the replay still needs to go through the instrumentation with high overhead at the "delayed" time. We aim for a more efficient fine-grained forensic auditing of replay executions.

Instead of solely relying on the replay to reconstruct every program state, we aim to record the needed events for auditing, which enables faster recreation of program states at the replay time. More generally, we explore an "analysis-aware" record and replay system that records certain data according to the analysis (e.g., recording data flow only).

We came up with two solutions for this goal. First, for a data-flow-analysis only record-replay, we record all the data flow at the program runtime so we can guarantee recreating it, which is sufficient for most popular forensic analyses. Second, we explore a custom recording environment that focuses on an efficient transformation from replay state to record state based on the specific analysis.

We tried to use a new Intel CPU feature for the first solution, i.e., Intel Process trace and process trace write [14]. Unfortunately, this trial is not successful. Though these features impose very little overhead, they tend to have data losses when being used in the scenario such as a data flow recorder. Coupled with program throttling, the data loss can be relieved but it adds 50%-200% overhead when running the standard benchmark. Currently, we are turning to explore the second solution which customizes the recorder by transforming the replay state to record state. As a hybrid approach, it records both the inputs and program states directly, which works as a combination for providing the data for the analysis and thus has the potential to achieve high efficiency in the replayed execution.

1.6 Thesis Roadmap

We will present how RAIN implaces the refinement to enable efficient analysis on system-wide executions in §3, and how RTAG extends to handle the data dependencies for networked systems in §4. Then we present how these systems worked and performed in the adversarial engagements by DARPA in §5; we finally present our exploration on a new record replay system for a more efficient replay and analysis in §6.

CHAPTER 2
RELATED WORK

Previous works in system monitoring and auditing strive to either ensure a low-overhead program runtime, or a fine-grained analysis having access to the memory space of the program. Unfortunately, none of them achieves both. Researches in record replay systems enables an decoupled analysis from the program runtime but they are either constrained in a single program or a massive whole-system replay including the operating system. None of these previous systems fits the goals of refinable forensic analysis. This thesis focuses on pursuing a fine-grained analysis at memory level, an efficient program runtime, and a significantly more efficient analysis suitable for system-wide auditing workload. This chapter iterates over the previous works in different categories.

2.1 System-call-level Analysis

Most attack detection systems that rely on system-call-level events track the program's interaction with the operating system. Such systems can detect malicious behaviors that particularly leave a trace of file/network IO operations. The performance overhead of such systems is lower than 10% such as DTrace [15], ProTracer [16]. Many of these systems (e.g., [15, 17, 10, 18]) have been accepted in the industry for system monitoring. For example, Linux supports the Linux Audit system [18], which records information about system events. PASS [19, 20] is a storage system that automatically logs and maintains provenance data. SPADE [21] is a cross-platform system that logs provenance data across distributed systems, and Linux provenance modules (LPM) [22] is a generic framework that allows developers to write Linux security module (LSM)-like modules that define custom provenance rules. In addition, ProTracer [16] is a lightweight provenance tracking system that supports system event logging and unit-level taint propagation, which is based

on BEEP [23]. However, none of the systems provides the instruction-level fine-grained provenance data that RAIN provides because they cannot achieve one of their main goals—minimizing runtime overhead—should they provide instruction-level provenance data.

2.2 Memory-level Program Analysis

Many advanced attacks involve fileless movements or lateral movements via IPC without explicit system call invocations, which escape the detection by the above systems. For example, the well known ssh injection attack [24, 25] relies on an integer overflow bug to achieve remote code execution. No system call event exists for the corruption in the memory, since it is a pure userspace operation without any system interaction. Due to the fact that a great number of exploits rely on memory corruption bugs such as use-after-free [26] and integer overflow [27], it is not sufficient to learn how the memory corruption occurs with the exploit payload just by analyzing the interactions between the program and the operating system [17, 10, 15]. To reason about the root cause and the vulnerability of the program, memory-level userspace instrumentation is demanded for analyzing memory states of the program even at single instruction step running with the exploit payload.

Though several dynamic binary instrumentation tools (e.g., Intel Pin [28], DynamoRIO [29]) are able to achieve this, they themselves incur high overhead (30%–100%) to the runtime of program. Dynamic taint analysis [30, 31, 32, 33, 34, 35] is a well-known technique for tracking the data flow from a source to a sink. Taint analysis is useful for runtime security policy enforcement [30, 31], malware analysis [34], and privacy leakage detection [35]. However, because of its excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [33], is six times as high), it is difficult to use it in a general computing environment. To solve this performance problem, several studies have proposed decoupled taint analysis techniques [11, 12, 36, 37]. The purpose of these techniques is to run a target process with a CPU core while performing taint analysis for a process with other idle CPU cores. Apparently this cannot be adopted in practice. Hence, researchers

are motivated to explore methods with the same level instrumentation while having low overhead at runtime.

2.3 Record-Replay-Based Decoupled Analysis

Unlike the deterministic replay technique that faithfully replays the previous execution (e.g., in [38, 39, 40, 41]), the replay-based decoupled analysis technique enables sophisticated analysis during replay. Arnold [9] is a state-of-the-art record-and-replay system supporting decoupled analysis during replay. The two main advantages of Arnold over two similar systems, Aftersight [42] and PANDA [43], are 1) its minimal recording overhead and 2) its process-group-wise replay with Intel Pin. These advantages stem from the implementation of Arnold inside the Linux kernel. By contrast, Aftersight is based on both a VMWare hypervisor (record) and QEMU (replay), and PANDA is purely based on QEMU. The main disadvantage of Arnold is that, unlike the other two systems, it is built inside the kernel, so it cannot record and replay the execution of the kernel.

As we have shown, these previous works achieves some goals of a good system for forensic analysis, but none of them has achieved all of the goals. Our systems enjoy the strengths of the previous systems, showing a system with accurate memory-level analysis, efficient program runtime, and a significantly more efficient analysis.

CHAPTER 3
REFINABLE ATTACK INVESTIGATION

3.1 Introduction

Since modern, advanced attacks are sophisticated and stealthy, collecting and analyzing attack provenance data has become essential for intrusion detection and forensic investigation. For example, many attack investigation or provenance systems monitor and log interesting system events continuously to identify which process interacted with an unknown remote host and which process accessed or modified sensitive files. If the systems find such a suspicious process, they will analyze its previous behaviors to determine whether it was attacked and which resources were affected by it.

Attack investigation systems, however, entail a practical limitation because of their two most important but conflicting goals—collecting a detailed log and minimizing runtime overhead. To ensure an accurate attack investigation, an instruction-level log would ideally record the execution of all of the CPU instructions of all programs. Nevertheless, such systems [44, 45, 34] also incur tremendous runtime overhead (4×–20×), so they are impractical in real computing environments. Therefore, as many attacks eventually need to use system calls to access sensitive resources and devices, other practical systems [22, 21, 23, 16] mainly focus on system-call information, the collection of which incurs low runtime overhead (below 10%).

Although system-call-based investigation systems are practical, they suffer from dependency ambiguity and explosion [23] because it is difficult to reconstruct accurate attack causality with only system-call information. For example, when a process reads from a number of sensitive files and sends some (encrypted) data to a remote host, knowing which sensitive files the process sends (or it might not send any sensitive data) without instruction-

or memory-level data-flow tracking that system-call-level log cannot provide becomes a challenge. To overcome this limitation, several systems [23, 22, 46, 16] instrument monitored programs to obtain interesting program-level information by modifying their source code or rewriting their binary code. Nevertheless, this approach is not scalable; that is, it must instrument each program again whenever it is updated. More importantly, it cannot cover dynamic code execution (e.g., code injection, self-modifying code, and return-oriented programming), which is frequently used by exploits.

We propose RAIN, a practical **R**efinable **A**ttack **IN**vesti-gation system, that selectively provides an instruction-level detailed log while minimizing runtime overhead. RAIN satisfies these conflicting goals using a *system-call-level record-and-replay technology* and *on-demand dynamic information flow tracking (DIFT)*. RAIN continuously monitors and logs system-call events and additional data for later replay while constructing a logical provenance graph. When it detects any anomalous event in the graph, it performs replay-based DIFT from the event to prune out any unwanted dependency. Performing DIFT for every process in the graph, however, is infeasible because the overhead of DIFT is too high (usually around 10×–20× and at best, 2.7×, using decoupling techniques [11, 12]). Instead, RAIN performs system call-level *reachability analysis* to extract a subgraph tightly related to the anomaly and then conducts DIFT only for processes belonging to the subgraph.

We evaluated RAIN using the red team exercises produced by the DARPA Transparent Computing program [47] and a recent real attack (StrongPity). These cases include normal background traffic with complex programs such as the Firefox web browser. Evaluation results show that RAIN is able to capture fine-grained causalities that accurately uncover the behaviors and effects of attacks, and in most cases the false positive rate is negligible. The runtime overhead is as low as 3.22% on SPEC CPU 2006, unlike previous instruction-level investigation approaches [44, 45, 34] whose runtime overhead is 4×–20×. Further, RAIN effectively reduces the number of processes to be replayed with DIFT and filters out, on average, more than 90% of processes. This is a considerable improvement in performance

over the previous approach, which has to replay the entire system and conduct DIFT against *all* processes.

Motivating Example. To illustrate the constraint of existing provenance systems and the contribution of our work, we refer to a recent attack called *StrongPity* [48]. The attack infected over 1,000 systems in Italy and several other European countries in late 2016. The purpose of StrongPity was to steal and tamper with the victims' data by means of compromised data transfer or archiving tools. Take, for example, Alice, a finance manager who maintains and manages contracts and bidding files. Alice usually uses a popular `ftp` extension called `FireFTP` in her Firefox browser to transfer files to other hosts, such as a machine hosting the shared folder for her team. In the first step of the attack, the extension in Alice's Firefox is upgraded to one that contains a backdoor, resulting from the distribution site of `FireFTP`, which has been compromised. A malicious extension accesses Alice's file system, collects data from certain files, and sends the data to an attacker's controlled site. In addition, the extension modifies certain incoming files before they are saved which also pollutes files that rely on the modified ones.

As we mentioned earlier, conventional system call-level tracing and auditing cause false positives in damage assessments when the source of the program (`Firefox` and `FireFTP` extension) is compromised and source instrumentation (if any) becomes untrustworthy. For example, the system call traces in Figure 3.1 indicate data leakage by connecting any `read` system call from sensitive files to the `send` system call directed to the malicious site. Many of these flows may be spurious if the user-space browser does not actually propagate the data from the file to the remote host (i.e., not all of the files being read are actually leaked). Similarly, many of the processes and files indirectly affected by interactions with the tampered file may not actually be affected at all. One needs to track the user-space data flow to precisely identify these dependencies.

With RAIN, after discovering that a host is controlled by an attacker, an investigator performs an upstream analysis originating from the host. With data pruning and selective

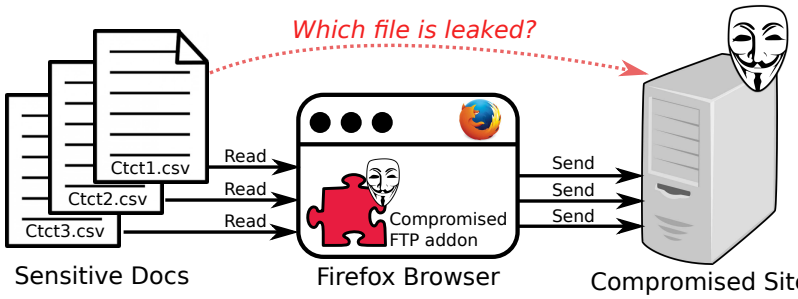


Figure 3.1: Example of causality inaccuracy (i.e., dependency explosion) in system-call-level provenance data.

DIFT, RAIN returns a provenance subgraph that contains the exact data leakages to the host. Although the malicious extension reads a number of files, it leaks only a small portion of them. By providing accurate analysis, RAIN saves the company from the fear of a large scale data leakage. We will revisit this example and elaborate on the details of the analysis in the following sections.

We summarize our contributions as follows:

- **A refinable attack investigation system.** We propose a new attack investigation system that efficiently records system-wide events in terms of system calls during runtime and refines the log with DIFT during replay to recover fine-grained causality. RAIN satisfies two conflicting yet important requirements: low runtime overhead and fine-grained causality information (at the CPU instruction level), both essential in the forensic analysis of attacks.

- **On-demand inter-process DIFT.** Instead of applying DIFT to whole-system events [44, 45, 34] which introduces tremendous overhead or which is likely infeasible, we introduce graph-based reachability analysis to filter out unrelated processes and selectively perform DIFT which makes inter-process DIFT feasible for attack investigation.

- **Accurate and comprehensive attack investigation.** We improve the accuracy of object-object, object-process, process-process causalities (§3.5.2) and significantly reduce the false positive rates generated by previous systems.

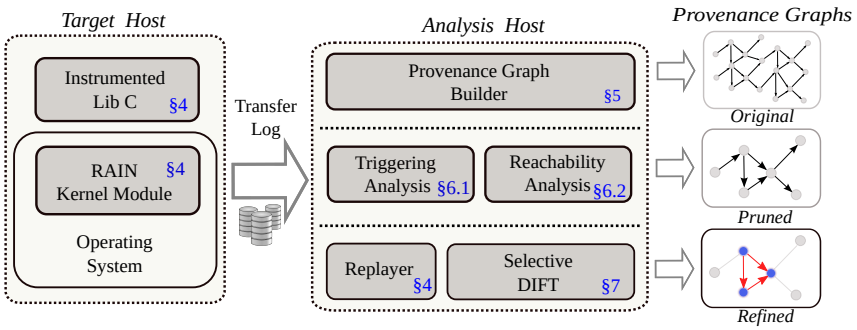


Figure 3.2: Overview of RAIN architecture.

The rest of this chapter is organized as follows: §3.2 describes our threat model. §3.3 provides an overview and describes the architecture of RAIN. §3.4 describes system logging and record-replay techniques, and §3.5 explains the provenance graph. §3.6 presents the reachability analysis and the process of identifying triggers. §3.7 describes how RAIN performs selective DIFT, and §3.8 summarizes its implementation and presents the results of evaluation. §3.9 discusses limitations and future work, §3.10 summarizes related work, and §3.11 concludes.

3.2 Threat Model and Assumptions

Our threat model is similar to those proposed in previous system provenance studies [22, 16, 19]—an OS and monitoring system are a trusted computing base (TCB). We take, for instance, an attacker who tries to attack the applications and resources of a system protected by RAIN and whose main goal is to exfiltrate sensitive data kept in the system or manipulate it to propagate misinformation. To achieve this goal the attacker may install malware on the system, exploit a running process, or inject a backdoor.

To realize a practical, refinable attack investigation system we assume the following: First, we assume that all of the attacks against the system begin after RAIN is deployed—that is, RAIN begins recording all of the attacks from their inception. Hardware trojans and OS backdoors are out of scope of this work. Second, we assume that although an attacker could compromise the OS or RAIN itself, the attacker has no way of manipulating the previous

provenance data containing attack attempts on the OS or RAIN. That is, although we can see the attacker attempting to compromise the OS or RAIN, any data recorded after a successful attack may not be reliable. In the future, RAIN could ensure data integrity by using previous secure provenance logging techniques [49, 50] and managing the provenance data in a remote analysis server. Also, by using state-of-the-art integrity-checking mechanisms [51, 52, 53, 54, 55], RAIN could determine when such an incident has occurred. Another assumption that we make is that an attacker uses only explicit attack channels, not side and covert channels which are beyond the scope of this work. Although RAIN does not yet have a solution stopping these attacks, we believe a record-and-replay approach has the potential to detect attacks as shown in [56, 57].

Note that although some instruction-level attack investigation systems [34, 44] are capable of detecting attacks against an OS, they are too slow to be used in a real computing environment and are mainly applied for in-depth malware analysis, running a small number of samples in a controlled environment. Thus, we only assume integrity-checking mechanisms in this work.

3.3 Overview

This section presents an overview of RAIN, a record-and-replay-based system that efficiently logs the whole-system events during runtime and conducts DIFT during replay to accurately determine fine-grained causal relationships between processes and objects (e.g., files and network endpoints) created during the execution of user-level processes.

Figure 3.2 represents the architecture of RAIN, which consists of two main components: the target host and the analysis host. In the target host, RAIN's kernel module logs all system calls that user-level processes have requested, including the return values and parameters that RAIN will use to generate a provenance graph. RAIN also records the execution of user-level processes by using kernel modules and an instrumented `libc` library to replay the processes later on. It collects all necessary information to reproduce the complete architectural

state of user-level processes (i.e., all non-deterministic values including random numbers). The target host then sends the system call and record logs to the analysis host (§3.4). In the analysis host, the provenance graph builder consumes the received system call log to construct a *coarse-grained* whole-system provenance graph that contains many security-insensitive causality events (§3.5). To refine the coarse-grained provenance graph, RAIN first detects *triggering points* representing suspicious events in the graph (e.g., accessing a sensitive file). Next, it initiates a *reachability analysis* (i.e., upstream, downstream, and point-to-point analyses) from the triggering points to create security-sensitive provenance subgraphs (SPS), which consist of basic units that replay with DIFT (§3.6). While selectively performing DIFT, the replay engine of RAIN replays each SPS to construct fine-grained causality subgraphs (§3.7). Lastly, with the fine-grained causality subgraphs, RAIN refines the original whole-system provenance graph to detect the true behavior and damages of any sophisticated attack that we were not able to observe in the original provenance graph.

3.4 Replay-able System Logging

3.4.1 System Logging

The system logging component resides inside the kernel of the operating system as a kernel module. We hook the system call table to intercept the arguments and return values of causality-related system calls. The component logs the semantics of system calls between kernel objects and events such as `open`, `read`, `write` file operations and `connect`, `recv`, `send` network operations. We also include essential semantics such as the file path and the file descriptor in the `open` syscall.

To uniquely identify the object, we log the related kernel semantics of processes and files, which include `inode`, `major`, `minor`, `gen` for files, and `pid`, `tgid` for processes. We also refer to kernel data structures if necessary (e.g., to get the string of a file's path from the file system structure, `dentry`) which enables us to reduce the log size for constructing the provenance graph to focus on unique processes and objects. We use the `relayfs` ring

buffer to efficiently transfer the system call logs from the kernel to the user space. The logs are packed, compressed, and transmitted off of the target host to a security-assured analysis host.

3.4.2 Enabling Replay-Ability

Compared to previous system logging schemes [16, 22, 58, 18], RAIN not only logs semantics for building the coarse-grained provenance graph, but also the non-determinism that enables faithful replay. For this purpose, we reuse Arnold, the open-source framework of a process-level record replay technique [9]. As an advantage, Arnold supports the independent replay of processes so we do not have to replay the entire system's execution (e.g., [43, 44]) for analysis. RAIN extends Arnold so that it supports *system-wide* recording, which accounts for every process execution in user-space. As a result, RAIN can replay *any* part of the system's execution on-demand for selective DIFT without losing completeness.

To replay the execution faithfully, Arnold records the return value of system calls, IPC communications such as signal and system V queues, and caches the data for every file or network I/O system call. For multi-threading applications, the `pthread` library in `libc` is hooked to record and enforce the order of the thread switch. To handle shared memory, Arnold replays involved processes cooperatively to regenerate shared data, and to improve the replay reliability. It also records and replays random numbers as well as `RDTSC` by using `prctl(PR_SET_TSC,PR_TSC_SIGSEGV)` from [59].

RAIN aims to detect and analyze attacks that may have previously gone undetected, so instead of being confined to a predefined list of known programs, Arnold's process level record-replay technique has been extended from a single process to *system-wide* executions. Although RAIN does not replay all of the recorded executions for refining attack provenance owing to the reachability analysis (§3.6) and selective DIFT (§3.7), this *system-wide* feature is critical as it enables us to refine *any* demanded part of execution. We achieve this by hooking the `execve` syscall inside of the kernel so that when a program is

loaded via `execve`, we force it to become a recording process by creating the recording contexts. Throughout the analysis RAIN can replay each demanded execution independently to resolve the associated fine-grained causality.

3.4.3 Storage Footprint

The storage use of RAIN comes from system logging and recording. We serialize the logging data using the Apache Avro [60] binary format, which incurs around 500MB–2GB per day for desktop use in our experiments. On the side of recording, the file, network I/O cache constitutes most of the storage cost. To optimize storage use, Arnold [9] applies compression, caches the data using "Copy-on-RAW", and manages the data pieces in a B-tree. In our experiment, the record logs of system-wide executions (excluding the OS) on a desktop produce around 2GB of storage per day. Therefore, the storage cost is 2.5–4GB per day (or less than 1.5TB per year). With the market price for a 2TB hard drive or cloud storage being around 50 US dollars, we believe that our storage cost is both reasonable and affordable. Note that instead of selectively storing data [16], we choose to store all of the raw data first and then generate the provenance graph selectively, following a set of pruning algorithms (§3.6). Our storage footprint reflects the size of raw data.

3.5 Provenance Graph

We construct a graph structure called *Provenance Graph* which contains the whole-system execution during the entire period of monitoring. RAIN uses this graph as the basic model to represent system objects, events, and their causal relationships. We begin by processing the syscall logs. When first constructed from system logging, the graph is coarse-grained. It is then pruned and refined incrementally according to analysis requests. We use nodes to represent system objects and edges to represent causality between system objects.

Table 3.1: Granularity of analysis.

Causality	Granularity
Process-Object	Coarse level
Object-Process	Coarse + Fine level
Process-Process	Coarse + Fine level
Object-Object	Fine level

3.5.1 Nodes

Nodes in the provenance graph can be classified into two categories: processes and objects. Processes represent user-level processes. Objects represent files and network endpoints. All nodes contain a timestamp that represents the time that the entity (represented by the node) was generated by the operating system. Each process node contains `pid`, `tid`, and process name. Each File node contains the full path name of the file and `inode`, `major`, `minor`, and `gen` which uniquely identify a file. Nodes representing network endpoints contain the IP address and port of the network entity. Note that the tracking scope of RAIN does not extend beyond the target host. In other words, we treat the remote host as a black box and do not track its internal logic or state. This limitation can be addressed if we apply RAIN on the remote host and causally relate the two hosts, but we will explore this issue in future work.

3.5.2 Causality Edges

Edges represent causal relationships between nodes in the provenance graph. We define four types of edges: process-object causality, object-object causality, object-process causality, and process-process causality. Among these causalities, we observe that only the process-object causality can be tracked reliably by syscall-level logging. The remaining causalities require either full or partial fine-grained user-space tracking. We summarize these granularity requirements in Table 3.1.

Process-Object Causality. Process-object causality, which denotes the causal relationship between a process and an object, is established when a user-level program accesses a file or network object. As the operating system provides these I/O services to user-space, this

causality can be captured by syscall logging (i.e., file or network I/O) without false negatives. For example, the data can be loaded from a file to the memory of process via a `read` syscall or written to a file from the process via a `write` syscall. In the case of a `read` syscall, the process has a directional edge to a file; and the case of a `write` syscall, a file node has an edge to a process. Similar causalities exist between processes and network endpoints. In the case of a `mmap` syscall, the direction of causality is determined by the syscall arguments such as `PROT` and `FLAG`.

Object-Process Causality. Object-process causality is established when the object affects the execution of a process or its control flow. Usually the process has an edge from an executable file if that file is loaded and executed by the process. RAIN captures this causality by monitoring the `execve` syscall. The use of libraries is typically tracked by `open`, `mmap`, or `dlopen` syscalls. However, this causality may not be true by just analyzing syscalls. For example, the developer may include libraries but not use them. To affirm causality between processes and libraries, we need to further track the control flow to see if its value (address) exists within a library's address space such as that described in [30]. Particularly for sophisticated attacks, an accurate object-process causality is crucial for detecting control-flow hijacking and identifying the sources of exploit payloads that could be used to access library functions. We accurately determine this type of causality during the DIFT phase.

Object-Object Causality. Object-object causality occurs in the case of data flow between two objects. The data flow inside of a process starts from an *inbound* object (e.g., via `read` syscall) and ends at an *outbound* object (e.g., via `write` syscall). One can infer this type of causality by simply pairing inbound and outbound objects. However, simply monitoring file or network I/O syscalls (e.g., our motivating example in §3.1) or statically analyzing on the program is inaccurate because we need to track the data propagation in the user-level execution. As the dynamic taint analysis is prohibitively expensive, RAIN tracks the Object-object causalities during replay (§3.7).

Process-Process Causality. Process-process causality is based on the relationships be-

tween two processes. Processes can be causally related if one is cloned by the other via the `clone` syscall, or they could be related because of inter-process communications (IPC). Some IPCs (e.g., pipe, message queues, and semaphores) can be observed from syscalls. However, causality cannot be accurately determined in the case of shared memory. Even though a `mmap` or `shmget` syscall indicates the establishment of an IPC channel, it does not necessarily mean data was actually exchanged between processes. To track such causality, RAIN relies on DIFT to monitor data propagation among memory operations.

3.5.3 Graph Construction

We construct the provenance graph by linking nodes and edges according to the above causality definitions. Our construction is based on uniquely identifiable objects. Since `pid` and `inode` are recycled if a process is terminated or file is deleted respectively, we use `path` as a unique identifier of processes and files (as nodes) since the collision of these objects with the same name is low in practice. After being processed from system call logs, each entry of a node or an edge is compressed and stored in a binary format. When requested for analysis, those within in the time frame are converted and imported into a graph database. In particular, we used Neo4j [61].

Semantic-Preserving Aggregation. Naturally, edges from I/O events such as `read` and `write` constitute a large portion of the graph. However, many are called successively indicating a single "large" `read` or `write` syscall execution. Therefore, we aggregate these successive calls for conciseness as inspired by [62]. For example, we merge two `read` syscalls as long as no other file system call occurs between them (e.g., a "write" to the same file). Note that we collapse these successive syscalls in an "indexing" style so that we do not remove the unique semantics of each individual edge. Thus, the selective DIFT still has the flexibility to perform taint tracking between desired I/O syscalls. The aggregation alleviates the traversing and storage costs of edges by 10%–50%.

3.6 Coarse-Level Pruning

After constructing the coarse-grained provenance graph, we prune it to generate a *security-sensitive provenance subgraph* (i.e., SPS) in two steps: a triggering analysis and a reachability analysis that uses the results of the triggering analysis. The SPS will be used as the target for selective DIFT, in which fine-grained causalities will be resolved.

3.6.1 Triggering Pinpoint

In the initial provenance analysis, we apply a set of methods to scan the logs and identify suspicious (i.e., *security-sensitive*) processes, events, and objects. The triggering analysis relies on three approaches: external signals, security policies, and customized comparisons. We perform the analysis offline by examining the provenance graph. This process can be done earlier when system call logs are available as with conventional intrusion detection systems [17, 63, 10].

External Signals. External signals are notifications from partners or third parties (e.g., an anti-virus company). For example, an analyst may receive advice from an anti-virus vendor to specifically check for the existence of certain executable files. All events performed by these executables can be labeled as triggering points. In our motivating example, the victim receives a notice that the distribution site of the FTP extension was compromised. This triggers an analysis of all behaviors of the browser starting from the malicious extension update.

Security Policy. Security policy checking also serves as a triggering pinpoint method. Based on administrative security policies, we create a set of policies that define *concerning* events used as *triggering points* for analysis. These policies include processes interacting with sensitive files or a sequence of events that deviates from the typical pattern of system calls. For example, it is a violation that a process reads from certain sensitive files and then sends read data to an unknown remote host. Recent research has shown that the detection

of attacks based on system-call sequence analysis can be improved with machine-learning techniques [64].

Customized Comparisons. We sometimes need to compare the states of objects at different times or locations to identify suspicious points. Take the data-tampering case in our motivating example. In order to identify files that have been tampered with, we compare the files (e.g., by comparing their hash digests) that have been downloaded via the browser extension to the original version of these files. If they differ, the *tampered* files are used as triggering points. This type of comparison typically requires application-specific semantics which are useful but not the focus of this work. For example, Gyrus [65] compares user interface (UI) inputs and network outbound traffic to determine user intention discrepancies.

3.6.2 Reachability Analysis

Starting from the identified triggers, we perform a reachability analysis to become aware of the *potential* original source(s) and impacts. This analysis prunes out unrelated executions and enables our DIFT to focus on resolving fine-grained causality in the attack-related executions. Although our DIFT is performed offline, the high cost of DIFT is not eliminated but *migrated* which is also pointed out by [11]. Hence we argue that it is still impractical to perform full taint tracking, even if it is performed offline. With reachability analysis, we set boundaries on the DIFT, avoiding tainting "dead" branches or regions of the graph.

The reachability analysis extends the triggering points to uncover possible upstream origins, downstream impacts, and causality paths between two points. Even though at this stage the graph includes only partial causalities (§3.5.2), computing the SPS on top of it and pinpointing the part that desires further DIFT is sufficient for capturing the the remaining causalities. With the SPS we can efficiently perform DIFT with a clear scope rather than the whole graph. We present the analysis interface in Algorithm 1 and the graph traversing algorithms in Algorithm 2.

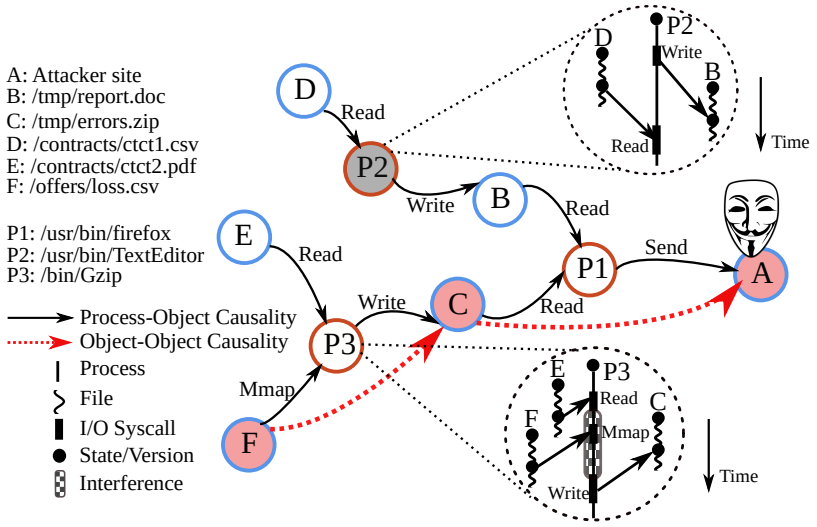


Figure 3.3: Coarse-level pruning and fine-level refinement on the motivating example.

Upstream and Downstream Pruning

Upstream pruning *reversely* scans the provenance graph from the triggering point and prunes out unrelated nodes and edges. The analysis follows the information flow and time sequence to extend the subgraph such that $Subject \rightarrow (write/send) \rightarrow Object \rightarrow (read/recv) \rightarrow Subject$. For example, in Figure 3.3, from the attacker's site (node "A"), we scan the `send` or `write` events to A; after finding Firefox (node "P1"), we further scan `read` or `recv` events that P1 performs earlier. For the shared memory case in process-process causality, we also scan for the syscall events that establish the IPC (e.g., `shmget` and `mmap`) and extend the SPS to that process. For downstream pruning we check forward events resulting from the suspicious processes and other files that are affected afterwards. Pruning also follows the information flow and extracts the downstream SPS. Pruning is either naturally bounded (meaning no more related causality is found) or bounded by the network interface. The gray shaded node indicates that process P2 is pruned from the SPS because of the negative interference between files B and D. The red-shaded nodes represent files on the causality path from origin F to attacker site A via file C.

Algorithm 1 Coarse Level Pruning Interface

```
function UpStrmPru(TPoint)
    GTraverse(TPoint,UP)
function DownStrmPru(TPoint)
    GTraverse(TPoint,DOWN)
function PtP(TPoint_1,TPoint_2)
    GTraverse(TPoint_1,DOWN)
    GTraverse(TPoint_2,UP)
    Mnodes ← FindMnodes(up_nodes,down_nodes)
    RecoverPaths(Mnodes)
```

Point-to-Point Pruning

Point-to-point analysis indicates *whether* and *how* two points are causally related in the graph. This analysis works on top of upstream and downstream pruning. Given two points, we first perform downstream pruning from the earlier point until the later point timestamp. Second, from the later point we perform upstream pruning until the timestamp of the early point. Then we inspect the two resulting subgraphs to identify the intersection set, called *meeting* nodes (FindMnodes() in Algorithm 2). Along with pruning, each point maintains the tags of its parent and ancestor nodes. Finally, we use the tags of meeting nodes to construct the full paths (i.e., SPS) (RecoverPaths() in Algorithm 2). The remaining causalities along the paths will be captured by the selective DIFT.

Data Interference in Memory Space

We further look into the system call sequence of each process execution. We observe that for object-to-object causality, the inbound object can be causally related to the outbound object only if their existence ranges overlap in the process memory space. We call these overlaps "interferences," which fortunately can be identified in the system call sequence. By examining interference situations, we skip performing DIFT in the case of non-interference. For example, in Figure 3.3, the outbound "report.doc" file (node B) has no interference with the inbound "contract1.csv" file (node D) because the `read` takes place after the `write`. Therefore, the interference analysis rules out the necessity of doing DIFT for the "TextEditor"

Algorithm 2 Information Flow Based Graph Pruning

```
Require: up_nodes, down_nodes
  function REGIST_INTERFERENCE(ne,na)
    if ne ∈ {read, recv} then
      if ne.timestamp ≤ na.timestamp then
        add_interference(ne,nd)
    else if ne ∈ {write, send} then
      if ne.timestamp ≥ na.timestamp then
        add_interference(ne,nd)
  function GTRAVERSE(TPoint,Direction)
    ngb_nodes ← read_graph(TPoint.uid)
    for node ∈ ngb_nodes do
      if Direction = UP then
        if TPoint.type = Process then
          if node ∉ {read, recv} then
            continue
        else if TPoint.type ∈ {File, Host} then
          if node ∉ {write, send} then
            continue
        up_nodes(node) ← TPoint
        regist_interference(node,TPoint)
        GTraverse(node,UP)
      else if Direction = DOWN then
        if TPoint.type = Process then
          if node ∉ {write, send} then
            continue
        else if TPoint.type ∈ {File, Host} then
          if node ∉ {read, recv} then
            continue
        down_nodes(node) ← TPoint
        regist_interference(TPoint,node)
        GTraverse(node,DOWN)
```

process (node P2). Meanwhile, the "Gzip" program (node P3) is an positive interference example, as both of the two upstream files ("contract2.pdf" (node E) and "loss.pdf" (node F)) once shared memory space with the outbound "errors.zip" file (node C).

It is this interference situation in the memory space that leads to possible data propagation (exchange between objects), which we later identify using DIFT. By identifying the exact interference situation of each process execution, we become aware of the part of the execution that requires fine-grained refinement as well as a source and a sink. In the DIFT, we fast-forward the replay to the start of interference (e.g., a `read` syscall), and then early-terminate at the sink. Each entry of interference includes the process, the first syscall that reads the inbound file, the last syscall that writes to the outbound file, the inbound file,

and the outbound file. To keep track of ordering, the timestamps of inbound and outbound syscalls are logged. As it is most effective when interference occurs at a late execution time or when it is short, we can skip most of DIFT. In §3.7 we show how to classify associated files with interference into groups so that one pass of DIFT is able to resolve all of the causalities in the group.

3.7 Selective Causality Refining

To further refine the graph and obtain fine-grained level causalities, we perform selective data tracking on top of the SPS. We re-compute (i.e., replay) user-space executions while performing taint analysis to determine causality in the interference cases (§3.6.2). Our approach entails tainting the bytes loaded to the memory space and tracking the propagation of the tainted bytes at the level of instruction execution. DIFT (or "taint analysis") has been implemented in previous work [32, 33, 11, 12]. We port open-source taint engines to develop our own taint engine that supports object-object, object-process, and process-process causalities (§3.7.1).

Because taint analysis is costly, we find that even offline analysis becomes impractical if we naively perform taint tracking on every process group. Therefore, we aim to minimize the cost of analysis by performing *directional taint tracking* in each group, orchestrating process groups for *information flow-based tainting* for upstream and downstream analyses, and reusing the taint results to avoid duplicate tainting. RAIN is able to track fine-grained causality along the *upstream* and *downstream* paths in the SPS according to the causality results in every branch. For each process group we locate the exact target of taint tracking according to the data interference of the presence of objects in the memory space (§3.6). We present how we conduct taint in every process group in §3.7.1 and how we orchestrate tracking across process groups in §3.7.2.

3.7.1 Directional Intra-process Tainting

According to the tainting targets determined in the SPS, RAIN performs DIFT starting from the source of interference (§3.6.2) and ending at the sink in every replay of the process execution. In our current prototype we determine causality to occur as long as any byte of the inbound object is propagated to the outbound object.

Interference Aggregation

Instead of running taint tracking on every pair of interferences, we aggregate them so they can be resolved in a single pass of taint tracking. This spares duplicate taints from propagating in the same execution trace. Aggregation takes place in the same process group. Suppose *n* interferences in the process group are related to the analysis request. We aggregate them by starting from the earliest interference and ending at the latest one. Then we run the taint tracking *one* time instead of *n* times. For example, in Figure 3.3, inside the P3 process, we aggregate the interference of files E and C via `read` and `write`, and files F and C via `mmap`, `write` because they belong to the same process P3 in the SPS. Thus we can resolve the causalities within them in one pass of tainting. When the tainting is performed it starts from the `read` syscall until the `write` syscall with the tagging of both E and F files as sources.

Replay and Taint Propagation

To allow taint tracking to work independently from a replayed execution, we adopt the analysis compensation technique from [66], which is able to differentiate the executions of Intel Pin from that of the program. First, no syscall made by Pin will be mixed by the recorded syscalls because the replayer is aware of their occurrences (i.e., it can differentiate between the two). Second, for memory space separation, the record log is scanned for any memory allocations. They will be allocated first so the replayed execution will not be affected by Pin.

Our DIFT engine is a set of Pin tools that reuses the open-source `libdft` [33], `Dytan` [32], and `dtracker` [45] projects for tracking object-object causality. The taint tags propagate on both data and control flow dependencies. Data dependency is tracked by monitoring the read/write memory operations at the instruction level between memories and registers, and control flow dependency comes from indirect branch dependency and incurs a higher overhead. We also implement tools that track object-process, process-process causalities. For cases in which object-process causality cannot be captured by the `execve` syscall, we taint track the data propagations and their impact on the control flow. We determine this causality when the return address or the `eip` register is tainted by the data of an input object (similar to [30]), which unveils typical memory corruption exploits that hijack the control flow. For the shared memory case in the process-process causality, we monitor shared memory-related syscalls (e.g., `shmget` and `mmap`) to map the shared memory among processes so that we are able to track the memory operation of a data transfer from a private memory space of one process (e.g., stack and heap) to the shared memory space, and then to the other processes. Additionally, we track the data propagation from an inbound object in one process to an outbound object in another process via shared memory.

3.7.2 Orchestrating Taintings Across Processes

In this section, we present how we perform taint tracking across processes according to the SPS. We regard taint tracking inside the process as a block function. To efficiently accomplish DIFT, we apply optimization techniques that minimize the tainting workloads. Specifically we introduce two methods of handling tainting in the *upstream* and *downstream* directions. Finally, we present how we refine causality paths for a point-to-point analysis by verifying the coarse-grained paths one-by-one and reusing the previous results cumulatively.

Downstream Refinement

Downstream refinement is capable of accurately identifying the impact of an attack, which is critical in both forensic analysis and intrusion recovery. Compared to conventional intrusion recovery approaches (e.g., Retro [58] and Dare [67]), RAIN produces accurate causality between involved files so the recovery can be performed only in files with true causality which eliminates false positives; otherwise innocent processes will be "re-executed."

Recall that when generating the SPS, RAIN also produces a pool of interference entries (§3.6.2) in which potential causalities exist. Starting from a designated point (e.g., a file), we identify the process and interference related to this point and then resolve the fine-grained causality. In the case of object-object causality, we run DIFT on the associated process and determine the outbound object(s) with true causality. From that object, we repeat the procedure to determine further causally related downstream objects bounded by the SPSs.

Take, for example, the data-tampering case in the motivating attack. The SPS reports that the tampered spreadsheet file "agreement.csv" has been later read by the auto-budget script which produces the budget and production plan files. More interestingly, the budget file is then used by the document editor which generates a season report. The triggering point in this case is the tampered spreadsheet file. Further interference entries with the file as *inbound* object will be pinpointed and taint tracking will be performed. We consider this interference situation an entry. Then we conduct taint analysis on the first (closest hop) process to identify the true outbound object and move further downstream making the found outbound object inbound object. As a result, we are able to repeatedly identify the exact downstream causalities and insert them into the provenance graph.

Upstream Refinement

Upstream refinement also begins at the triggering point, but proceeds in the *reverse* order of the execution time. The SPS appears in an acyclic-directed graph shape with the latest point being the triggering point (e.g., the file leaked by the compromised FTP extension).

To identify the leaked file and its provenance, we locate the associated process in the SPS. The taint tracking on the replay of the process execution determines the real causal parents so the next rounds of taint tracking are performed only at these parent files. Taint tracking continues recursively until it hits a boundary advised by the SPS. At each branch where multiple inbound objects exist, refinement continues only on the *true* inbound object(s) and ensures that they are outbound objects for the next round of tainting.

In Figure 3.3, from the attacker's site, we begin running upstream refinement performing tainting at P1 (i.e., the Firefox session). We find that although the "report.doc" file (node B) and the "errors.zip" file (node C) are both inbound, only file C is causally related to the attacker's controlled host (node A). We drop file B and continue refinement on the branch from file C. Again we find that "Gzip" (node P3) has input files "contract2.pdf" (node E) and "loss.csv" (node F), but only file F exhibits causality with file C, so we continue along the F branch. As a result, we eliminate the unnecessary workload of tainting *dead* branches that do not reach the triggering point.

Extending Causality Across Processes. As an optional feature, we keep track of the causality across processes (e.g., in Figure 3.3, file F to site A). We maintain a *shadow tagging file* for each file that is accessed by more than one process. This tagging file keeps track of the source tag of every byte in the file so that RAIN can track the causality between two separate files in different processes. For example, in Figure 3.3, the shadow tagging file of file C is generated when tainting is performed on P3. The tagging file contains the bytes with causality between files E and F. When we replay and taint track P1, we refer to the tagging of file C and acknowledge that the leaked data includes both files C and F because part of the contents of C originated from file F. Accompanied by upstream refinement, RAIN constitutes the result of this Point-to-Upstream analysis that file C has been leaked to the attacker while certain leaked contents of C originate from F. In our current prototype, this feature is optional and on-demand as it incurs higher tainting overhead. Note that tracking the data with tags from previous objects requires full-length tainting from the inbound to

outbound objects.

Point-to-Point Refinement

Recall that causality may have been included within the SPS (§3.6.2). With the help of the SPS, heavy DIFT is applied to *verify* the data flow on the path where the fine-grained causalities (e.g., object-object) occur. This filters out many unrelated branches that would have incurred high analysis overhead.

Based on the processes on each path in the SPS, we replay and perform taint tracking on the process groups in the path to verify true causality. From the start to end points along the path, each process group is replayed and taint tracking performed on the specific interference between the inbound and outbound system calls. The inbound object is tagged and the running of the process propagates the tags and monitors whether it hits the outbound object. If it does, verification continues. Otherwise, it terminates and returns a negative result. The verification runs until an end point. If all interferences along the path are positive, we refer to the path as "causality positive." At the end, the refinement returns all the causality-positive paths.

The verification procedure is optimized by reusing the taint results for each group. In the implementation, we store the causality between specific inbound and outbound files in a database. In the remaining verifications, we start by searching the database for existing causality facts. Then we reuse them if possible without performing the same taint tracking again. Because we reuse the results, taint verification takes less time as we verify more paths. Particularly for the point-to-point case, the taint tracking in every process group is optionally run in parallel to accelerate the analysis if more computing resources are available. As every process group is independently recorded and independently replay-able, we tag every process group with a *symbolic* tag and will resolve the tag propagation with real tags. The use of more resources with less time consumption represents the cost of this optimization. In this case, refinement time decreases to the level of the longest DIFT.

Table 3.2: Implementation complexity.

Host	Module	LoC
Target host	Kernel Module	2,200 C (Diff)
	Trace Logistics	1,100 C
Analysis host	Provenance Graph	6,800 C++
	Trigger/Prune	1,100 Python
	Selective Refinement	900 Python
	DIFT Pin tools	3,500 C/C++ (Diff)

3.8 Implementation and Evaluation

We implemented a prototype of RAIN in Linux. In the kernel module of the target host, we implemented the system logging logic with comprehensive semantics to build the provenance graph and to support whole-system recording. On the analysis host, we implemented the construction of the provenance graph and trigger/prune methods as well as DIFT that support object-object, object-process, and process-process causalities on top of [33, 32, 45]. The complexity of implementation is summarized in Table 3.2. We plan to release the source code of RAIN.

Our evaluation addresses the following questions:

- How well does RAIN detect various attack scenarios (§3.8.1, §3.8.1)?
- How accurately does RAIN prune and refine provenance graphs (§3.8.1)?
- How much overhead associated with analysis, runtime, and storage does RAIN have (§3.8.2)?

In our evaluation environment, we set up the target and analysis host individually on two bare-metal machines both powered by Intel Xeon(R) CPU W3565 3.2GHz; the target host has an 8GB RAM and 512GB hard drive, and the analysis host has a 32GB RAM and 2TB SSD hard drive. They are connected by a 1GB Ethernet cable. Both machines run Ubuntu 12.04 LTS (i386).

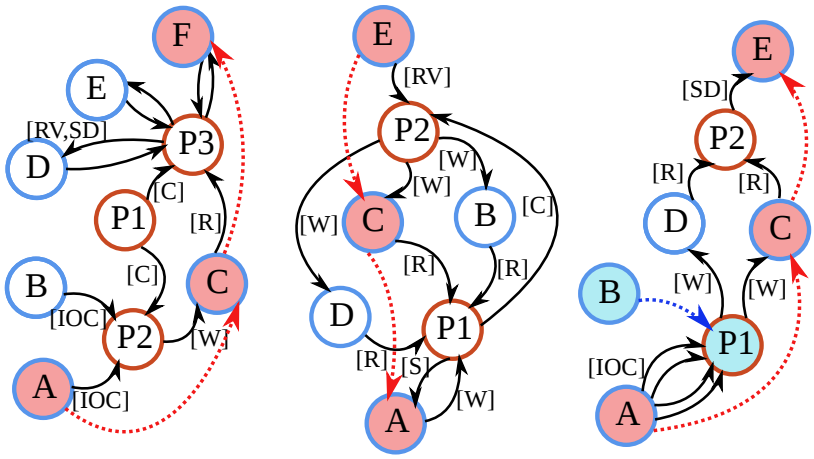
3.8.1 Security Analysis

Using various attack scenarios, including the motivating example, we evaluate the accuracy gains and conduct a set of red team exercises from the DARPA Transparent Computing program [47].

Motivating Example

We demonstrate the end-to-end procedure and efficacy of RAIN at detecting and analyzing the motivating attack example (§3.1). The attack exploits the `FireFtp` addon of Firefox to steal a user's data and tamper with downloaded files. At the triggering pinpoint phase of the analysis, the security team of the company was notified that an originally trustworthy site (e.g., white-listed in the firewall policy) was compromised for one week until they confirmed and recovered from the leakage of critical contract details (i.e., *External Signal* in §3.6.1). In addition, they received complaints from the audit team about abnormal changes in the numbers in a spreadsheet file when they compared the downloaded spreadsheet file to the original one stored in the archive (i.e., *Customized Comparison* in §3.6.1). Because of the dependency between documents, the team was also concerned about the impact on other files. With these triggering points, the security team, to accurately determine *what was stolen* and *what was affected* by the attack, queried RAIN for the exact causalities around the compromised site and suspicious files.

To find out what was leaked to the attacker's controlled site, RAIN performs a "Point-to-Upstream" analysis from the site to identify the leaked data. First, RAIN extracts the SPS from the provenance graph by pruning off unrelated nodes and edges and downsizes it to around 20% of the original provenance graph. Then RAIN performs refinement on some of of the process executions, including the Firefox process which communicates with the site. After performing selective refinement, RAIN determines that even though multiple files were accessed, only "ctct1.csv" was leaked. The refinement further reduces the size of the SPS to around 10%, which reveals the few but accurate causalities originating from the



(a) NetRecon (b) ScreenGrab (c) CameraGrab

Figure 3.4: Simplified provenance graph with highlighted accurate causality path

malicious site (Figure 3.3). RAIN also locates the set of files affected by the tampered file as it is used by a finance program to generate reports and other documents. The accurate results generated by RAIN ensure that the company is aware of the scope of data leakage and the impact of the data tampering without panicking or having to carry out unnecessary recovery efforts.

TC Red Team Exercise

We use the set of attack scenarios from the red team exercise of the Transparent Computing (TC) program [47] to continue our evaluation. The attack first installs an *implant-core* on the victim's system via social engineering (e.g., email). After installation, the implant-core communicates with the attacker's host (e.g., the C&C server) and receives and performs future attack tasks. We use four unit attack examples (i.e., NetRecon, ScreenGrab, CameraGrab, and AudioGrab) to demonstrate how RAIN works and what amount of accurate causality is generated.

Table 3.3: Incremental evolvement with pruning and refining.

Analysis Stages		Coarse Level Pruning			Fine Level Refinement					False Positive Rate		
Items		Nodes/Edges			Nodes/Edges		Paths			Coarse%	Fine%	REDUC%
Attacks	**Analysis**	**ProvGraph**	**SPS**	**Prune%**	**Result**	**Added%**	**SPS**	**Result**	**Added%**			
	A(O-Up)		3,024/26,749	15.4%/19.7%	342/2,621	11.3%/9.8%	-	-	-	**67.0%**	**0.0%**	**100.0%**
MotivExp	A(O-Dn)	19,634/135,474	1,822/13,981	9.3%/10.3%	46/336	2.5%/2.4%	-	-	-	**55.6%**	**0.0%**	**100.0%**
(3h02m)	A(O-O)		389/733	1.9%/0.5%	98/222	25.2%/20.2%	51	19	37.3%	**69.1%**	**0.0%**	**100.0%**
	A(O-Up)		2,394/17,691	18.5%/20.5%	198/210	8.3%/11.9%	-	-	-	**70.3%**	**23.4%**	**66.8%**
NetRecon	A(P-Dn)	12,892/86,376	1,234/8,880	9.6%/10.3%	86/799	7.7%/9.0%	-	-	-	**84.7%**	**13.0%**	**84.7%**
(2h38m)	A(O-O)		147/287	1.1%/0.3%	34/66	23.2%/23.0%	12	4	33.3%	**66.6%**	**0.0%**	**100.0%**
	A(P-Up)		1,348/9,189	18.4%/19.8%	156/952	8.2%/7.9%	-	-	-	**90.5%**	**0.0%**	**100.0%**
ScreenGrab	A(O-Dn)	7,327/46,367	895/4,877	12.2%/10.5%	72/351	8.1%/7.2%	-	-	-	**82.1%**	**0.0%**	**100.0%**
(1h13m)	A(O-O)		21/30	0.28%/0.07%	5/4	23.8%/13.3%	9	5	55.5%	**77.4%**	**0.0%**	**100.0%**
	A(P-Up)		1,603/11,102	30.2%/33.2%	96/477	6.0%/4.3%	-	-	-	**32.0%**	**0.0%**	**100.0%**
CameraGrab	A(O-Dn)	5,308/33,440	589/3,317	11.0%/9.9%	59/70	10.5%/2.1%	-	-	-	**29.8%**	**0.0%**	**100.0%**
(39m)	A(O-P)		101/268	1.9%/0.8%	24/59	24.1%/22.0%	9	7	77.7%	**44.2%**	**0.0%**	**100.0%**
	A(O-Up)		992/6,846	20.2%/20.5%	49/232	4.9%3.4%	-	-	-	**39.7%**	**0.0%**	**100.0%**
AudioGrab	A(P-Dn)	4,909/33,382	415/3,394	8.5%/10.1%	31/161	7.4%/4.7%	-	-	-	**48.2%**	**0.0%**	**100.0%**
(30m)	A(P-P)		230/1,392	4.7%/4.2%	84/519	36.5%/37.3%	22	18	81.8%	**29.3%**	**0.0%**	**100.0%**

Table 3.4: Analysis request details.

Schemes	Analysis	1st Point	2nd Point	Time (s)
MotivExp	A(O-Up)	AttkSvr	-	-
	A(O-Dn)	TmpedFile1	-	-
	A(O-O)	CtctA.csv	AttkSvr	305
NetRecon	A(O-Up)	NR.log	-	-
	A(P-Dn)	NetScan	-	-
	A(O-O)	AttkSvr	NR.log	94
ScreenGrab	A(P-Up)	ScrnGrab	-	-
	A(O-Dn)	X11Svr	-	-
	A(O-O)	X11Svr	AttkSvr	68
CameraGrab	A(P-Up)	ImpCore	-	-
	A(O-Dn)	CmrGrab(file)	-	-
	A(O-P)	CmrGrab(file)	ImpCore	38
AudioGrab	A(O-Up)	AttkSvr	-	-
	A(P-Dn)	Firefox	-	-
	A(P-P)	Firefox	RptGen	418

NetRecon. First, the implant-core clones a process called NetRecon which collects network configuration information that it saves to a temporary file. Second, the implant-core clones another process that scans neighboring hosts based on network configuration information. The triggering analysis finds suspicious collecting behavior by spotting a downloaded file conducting a series of `ioctl` requests `SIOCGIFHWADDR` and `SIOCGIFBRDADDR`. The results of the "point-to-downstream" analysis shows that the cloned process reads the temporary file and tries to connect other internal hosts that are determined by the temporary file. In addition, the point-to-point analysis between the "NetRecon.log" and neighboring hosts shows the effectiveness of RAIN involving control flow dependency. Figure 3.4(a) highlights the key causality between `eth0` and another neighbor host. It shows a simplified provenance graph with highlighted accurate causality path (red dotted line for object-object causality and blue dotted line for object-process causality; the nodes on the path are colored accordingly). Notations: (a) A/B: eth0/eth1; C: /tmp/netrecon.log; D/E/F: internal hosts; P1: ImplantCore; P2: NetReCon; P3: Scanner. (b) A: attacker server; B/C/D: screen.png; E: /tmp/.X11-unix/X0; P1: ImplantCore; P2: ScreenGrab. (c) A: /dev/video; B: malicious site that contains CameraGrab payload; C/D: pictures; E: attacker server; P1: Firefox browser; P2:

ImplantCore. Additionally, we perform other types of analyses and list the incremental results in Table 3.3. The **Nodes/Edges** column of provenance graph shows the whole system executions during this period; **Nodes/Edges** of SPS tells the size of pruned data in different queries; **Prune%** gives the portion of pruned data to the original one; In the fine-level refinement stage, we show the number of nodes and the number of added causality edges in **Nodes/Edges**, along with the **Added%** rate. Particularly for the "Point-to-Point" analysis, the **Paths** column tells the number of paths in the SPS and the numbers after refinement. Column **FP-Coarse%**, **FP-Fine%**, and **REDUC%** indicate the false positive rate with coarse-level provenance data, with RAIN refinement, and the reduction percentage.

ScreenGrab. The implant-core downloads a "ScreenGrab" program that occasionally captures the screenshot of the victim's desktop and selects certain shots to send to the attacker's server. While the attack occurs, the user performs various background desktop actions such as web browsing. Our triggering analysis learns that a site is controlled by the attacker. Starting from the malicious site, RAIN conducts an upstream analysis in order to identify a causal relationship with the triggering point. It begins by extracting the SPS from the provenance graph, and then performs a fine-grained analysis to refine the SPS to obtain an accurate causality subgraph. We can see the executable ScreenGrab has multiple inbound traffic from the X11 server (i.e., via Unix domain socket `/tmp/.X11-unix/X0`) and outbound traffic to a file. Then the implant-core sends this file to the attacker's host. RAIN is able to identify exactly which file is sent. We highlight the SPS with refined causality in Figure 3.4(b) in red.

CameraGrab and AudioGrab. The victim's Firefox browser is exploited with a zero-day exploit and its control-flow is hijacked to the CameraGrab and AudioGrab payloads. The exploited browser then uses a camera and a microphone to spy on the user's behavior and saves it in images and audio files respectively. Finally, the implant-core selects certain files and sends them to the attacker server. During this process, the user sometimes finds that the LED light on the camera is on despite having no intention of using the camera. The

Table 3.5: Analysis cost comparison.

		Time (s)			#Taint		
Attack	**Analysis**	**PruT**	**RefiT**	**T(P+R)**	**RAIN**	**None**	**Fraction%**
MotivExp	A(O-O)	759	2,321	3,080	34	720	**4.7%**
NetRecon	A(O-O)	140	1,320	1,460	13	138	**9.4%**
ScreenGrab	A(O-O)	127	253	380	5	99	**5.0%**
CameraGrab	A(O-P)	326	757	1,083	19	141	**13.4%**
AudioGrab	A(P-P)	301	687	988	11	310	**3.5%**

triggering point is `ioctl` syscalls which communicates with the device. To determine the root cause, we perform "point-to-upstream" analysis to check for the specific object-process causality that causes the exploitation of Firefox. The results (Figure 3.4(c)) indicate a causality between the CameraGrab payload and the browser as the instruction pointer of Firefox goes to the payload. A further check of Firefox reveals that the main page has become a malicious site, so the browser is exploited every time it is started.

Pruning and refinement

In general, the resulting subgraph is substantially smaller than the global graph (> 90%) as well as the SPS that is computed by the coarse-level analysis. More importantly, because of DIFT, the analysis reveals the *true* causalities. We analyzed several attack scenarios and summarize their incremental pruning and refinement results in Table 3.3 (the specifics of the analysis requests are listed in Table 3.4). In particular, we list the false positive rates using coarse-level data with RAIN refinement and the reduction ratio. The potential causalities (denominator) are counted according to the "dependency explosion" [23] definition in which each output is assumed to depend on all the earlier inputs. With RAIN, most false positives in the provenance graph are eliminated (i.e., a 100% reduction), but we also encountered two cases in which false positives remained after refinement. When we took a closer look at the DIFT, we observed the "over-tainting" situation that occurs during control flow-based propagation which is a known limitation of DIFT. In general, RAIN effectively improves the precision of attack investigation.

3.8.2 Performance

Analysis Performance

To fairly examine the time duration and tainting workload induced by RAIN, we evaluate the cost of analysis using bounded point-to-point queries. In Table 3.5, we first show the time duration for RAIN to prune (column **PruT**) and refine (**RefiT**) the data using the point-to-point refinement in parallel (i.e., the longest duration among instances of tainting). The **PruT** column lists the time used to extract the subgraph prior to fine-grained refinement. **RefiT** shows the time it takes to selectively refine causality using taint analysis; **T(P+R)** represents the total analysis time and **#Taint-RAIN** how many process groups are replayed and taint tracked with the point-to-point refinement algorithm (§3.7.2); **#Taint-None** provides the number of process groups to be replayed and tainted between the two time points without reachability and selective DIFT algorithms. The average fraction ratio is 5.8%.

We then evaluate the performance of the reachability analysis and selective DIFT. We first list the number of all of the process groups between the two points in the **None** column. If one attempted to refine the causalities without applying the pruning (§3.6) or the selective DIFT algorithms (§3.7), this number would represent the load. It would also reflect the user-land part of the taint workload in full-system DIFT systems (e.g., [44, 34]). The number of tainting instances that RAIN performs for the same task is listed in the **RAIN** column. We find that our algorithm is effective, significantly reducing the tainting workload to a fraction of 5.8% on average. Note that we focus on the factual tainting workload the analysis must take, rather than the total time. After all, one can parallelize the workload on multiple machines to reduce time consumption.

Runtime Performance

We evaluate the runtime performance of RAIN with SPEC CPU2006 benchmarks listed in Figure 3.5. The runtime overhead of only running system logging is represented by the

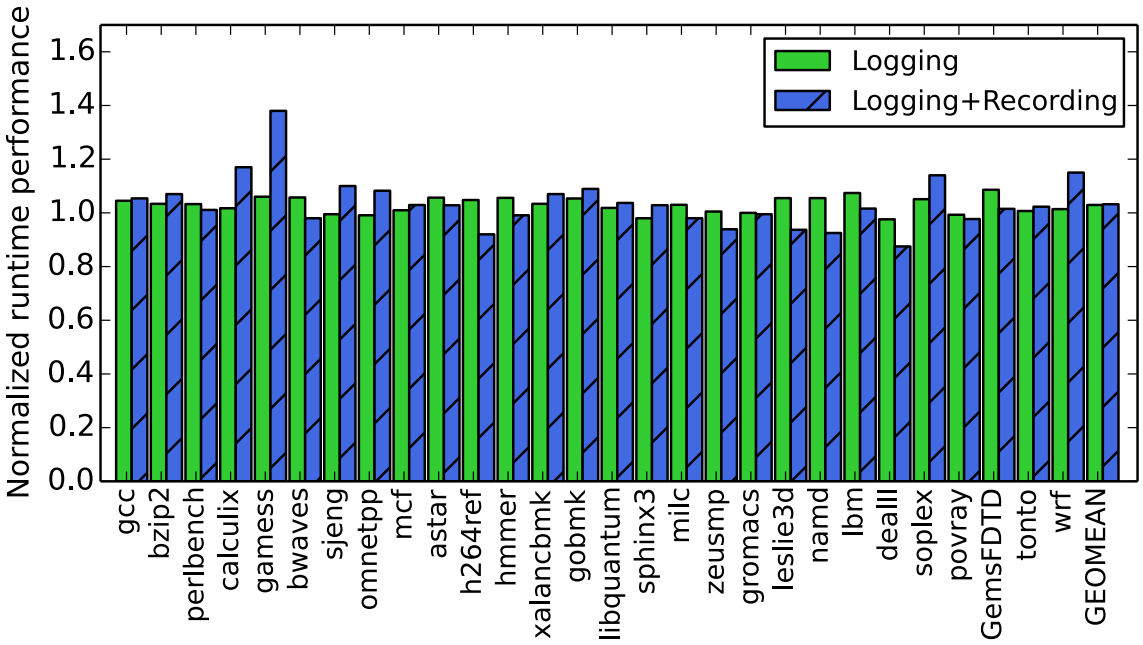


Figure 3.5: Normalized runtime performance with SPEC CPU 2006.

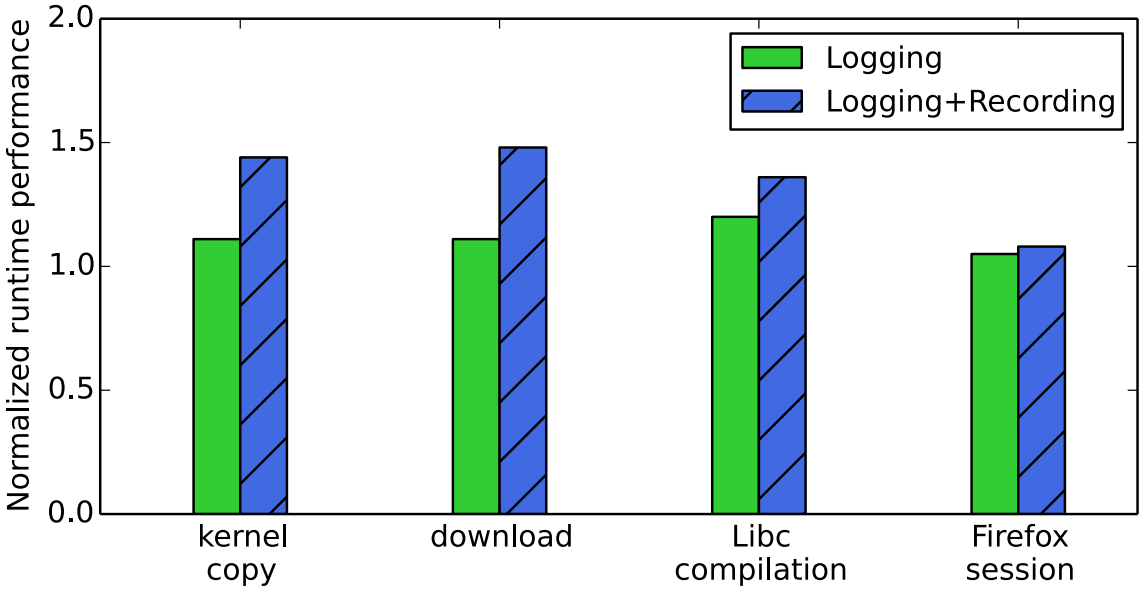


Figure 3.6: Normalized runtime performance with I/O intensive applications.

green bars for various testing items. The overhead of logging plus recording is listed in the blue bars. The geometric mean of runtime overhead in a logging+recording mode is 3.22%.

Besides the CPU intensive benchmark, we also run I/O intensive applications as RAIN hooks system calls and caches file or a network I/O. We compare the runtime performance of four applications: copying the Linux kernel 3.5.0 archive with cp, downloading a 450MB

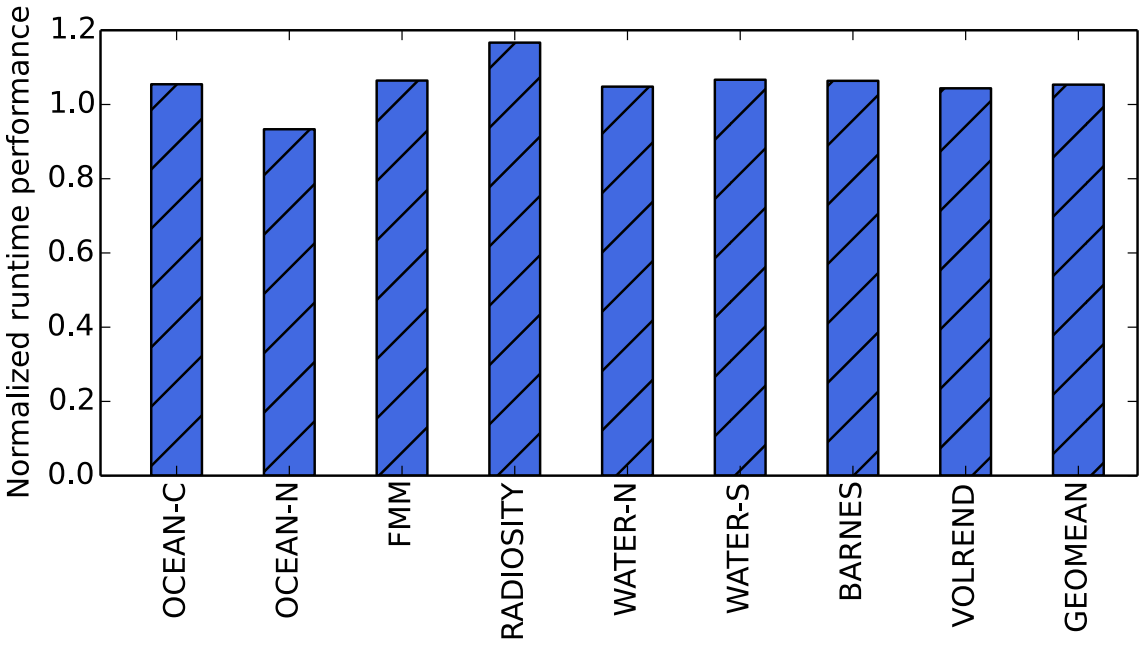


Figure 3.7: Normalized runtime performance (logging+recording) with SPLASH-3 multi-core benchmark (4-core CPU). OCEAN-C: contiguous; OCEAN-N: non-contiguous; WATER-N: nsquared; WATER-S: spatial.

Table 3.6: Storage cost (MB).

	Storage usage (MB)		
Case	**Log**	**Record**	**Total**
MotivExp	45.6	155	201
NetRecon	29	137.1	166
ScreenGrab	16.6	97.3	114
CameraGrab	15.8	89.2	105
AudioGrab	18.2	115.4	133
Libc compiliation	327	413	740

video `mp4` file from a local area network with `wget`, compiling the `eglibc-2.15` library, and loading `cnn.com` in Firefox. Figure 3.6 illustrates the normalized overhead breakdown in terms of system logging and full mode (logging+recording). In these I/O intensive cases, RAIN incurs no more than 50% overhead.

To evaluate the runtime performance of RAIN in multi-core machines, we run the SPLASH-3 [68] multi-core benchmark with a 4-core CPU and summarize the results in Figure 3.7. The geometric mean of runtime overhead (logging+recording) is 5.35%, and RAIN is able to faithfully replay all the benchmarks without divergence.

Storage Cost in Scenarios

We measure the storage cost of RAIN with the scenarios used in §3.8.1 and a high workload case (i.e., compiling `eglibc-2.15`), the results of which are summarized in Table 3.6. The compiled `libc` is around 235MB, which is smaller than either the system or record log. This is because RAIN not only cached the target files that were built but also the temporary files generated during the compiling. Even though the log size is larger than the resulting size of compilation, it shows RAIN's ability to re-construct (and analyze) a transient state of a complex program execution (we presented the storage footprint of daily usage in §3.4.3).

3.9 Limitations and Discussion

This section presents the limitations of RAIN and directions of future work. One limitation of RAIN is that it is a kernel-based system. It is able to record, replay, and analyze the activities of user-level processes but unable to monitor the kernel activities because it trusts the kernel. If the kernel is compromised, RAIN is no longer able to create reliable provenance data. Thus if such an incident ever occurred, RAIN could use kernel-integrity monitoring techniques [51, 52, 53, 54, 55] that detect and filter out misinformation. In the future, we will port RAIN to a hypervisor that records and replays kernel activities while reducing attack surfaces. This approach will allow RAIN to support commercial off-the-shelf (COTS) OSes (e.g., Windows). In addition, the semantic information of RAIN poses a limitation because it is less comprehensive than that of source-code-instrumentation-based approaches [23, 16] since it assumes no assistance from software developers (i.e., annotation). We believe that these approaches are orthogonal: while RAIN is successful at extracting fine-grained information from COTS programs or unknown binaries (e.g., malware), instrumentation-based approaches are effective at collecting semantically rich information from supportive programs. Another limitation is the over-tainting issue that we encountered, particularly when dealing with control flow-based propagation. This problem, which has always plagued

Table 3.7: Comparison of full-system provenance systems.

DIFT Systems	Cross Host	Inst Time	Tag Dep	Run Over	DIFT Over(T/M)
Dytan [32]	×	Runtime	Inlined	High	High/High
DataTracker [45]	×	Runtime	Inlined	High	High/High
Panorama [34]	×	Runtime	Inlined	High	High/High
ShadowReplica [11]	×	Runtime	Inlined	High	Low/High
Taintpipe [12]	×	Runtime	Inlined	High	Low/High
Panda [43, 69]	×	Replay	Inlined	High	High/High
Arnold [9]	×	Replay	Inlined	Low	High/High
RAIN	×	Replay	Inlined	Low	High/High
Jetstream [37]	×	Replay	Inlined	Low	Low/High
TaintExchange [70]	✓	Runtime	Inlined	High	High/High
Cloudfence [71]	✓	Runtime	Inlined	High	High/High
RTAG	✓	**Replay**	**Decoupled**	**Low**	**Low/Low**

DIFT approaches needs to be addressed. Since RAIN relies on triggering analysis to initiate a fine-grained analysis, it could either miss or delay the detection of some stealthier attacks Further, faulty triggers could simply waste the time and energy resources of RAIN. To solve this problem, we plan to develop an anomaly-based self-triggering mechanism that automatically initiates a fine-grained analysis. Lastly, the storage overhead of RAIN is greater than that of other systems such as ProTracer [16]. Unlike such systems, RAIN records all kinds of system calls (§3.4.3) to support replay-able execution, so the additional storage overhead appears to be unavoidable. We plan to explore a further reduction of storage overhead, for example, by compression and deduplication.

3.10 Related Work

Full-system Provenance Logging. Full-system provenance logging is essential to detecting complicated attacks. For example, Linux supports the Linux Audit system [18], which records information about system events. PASS [19, 20] is a storage system that automatically logs and maintains provenance data. SPADE [21] is a cross-platform system that logs provenance data across distributed systems, and Linux provenance modules (LPM) [22] is a generic framework that allows developers to write Linux security module (LSM)-like modules that define custom provenance rules. In addition, ProTracer [16] is a

lightweight provenance tracking system that supports system event logging and unit-level taint propagation, which is based on BEEP [23]. To reducing logging workload, ProTracer "taints" in order to keep track of the units, which fundamentally differs from the dynamic instrumentation-based taint tracking that we apply. However, none of the systems provides the instruction-level fine-grained provenance data that RAIN provides because they cannot achieve one of their main goals—minimizing runtime overhead—should they provide instruction-level provenance data. RecProv [72] relies on a user-level record and replay technique to recover syscall-level provenance, but its replay does not perform instruction-level instrumentation, so it provides no finer-grained causality. DataTracker [45] performs taint tracking and provides fine-grained causality data on individual files, but because of the high execution overhead of taint analysis, using it as an analysis system instead of a production system is impractical. PROV-Tracer [44] is built on top of PANDA [43], which can also provides instruction-level granularity. However, the QEMU emulator it bases on is around 5× slower than a native execution. We summarize the comparisons between RAIN and previous provenance systems in Table 3.7. We compare the existing systems and RAIN in terms of provenance granularity, runtime overhead and requirement. "Workflow" in the brackets is another mode that monitors user-land applications, but requires source code instrumentation. RAIN achieves both efficient runtime and instruction level analysis granularity while does not require source or binary instrumentation.

Network Provenance Systems. Network provenance systems [49, 50, 73, 74] focus on tracking network-level provenance between computing hosts belonging to the same distributed or enterprise network environment. Their main goal is to find faulty (or compromised) hosts that attempt to attack other hosts in the same network by monitoring and analyzing network traffic, and to ascertain how the faulty hosts were compromised. Since they focus on network traffic, system-level fine-grained provenance (e.g., CPU instructions that were executed) are beyond the scope of the work that proposed these systems. Thus, RAIN is orthogonal to these network provenance systems such that both can be

simultaneously used to fully cover intra- and inter-system provenance.

Replay-Based Decoupled Analysis. Unlike the deterministic replay technique that faithfully replays the previous execution (e.g., in [38, 39, 40, 41]), the replay-based decoupled analysis technique enables sophisticated analysis during replay. Arnold [9] is a state-of-the-art record-and-replay system supporting decoupled analysis during replay. The two main advantages of Arnold over two similar systems, Aftersight [42] and PANDA [43], are 1) its minimal recording overhead and 2) its process-group-wise replay with Intel Pin. These advantages stem from the implementation of Arnold inside the Linux kernel. By contrast, Aftersight is based on both a VMWare hypervisor (record) and QEMU (replay), and PANDA is purely based on QEMU. The main disadvantage of Arnold is that, unlike the other two systems, it is built inside the kernel, so it cannot record and replay the execution of the kernel.

We choose Arnold [9] as the base system of RAIN mainly because of its efficiency. Note that RAIN's functionalities (e.g., full-system recording, provenance data generation, reachability analysis, and refinable DIFT) are orthogonal to Arnold, so we can apply them to other systems easily. In fact, we have PANDA-based RAIN, which provides the same functionalities even though its recording overhead is excessive (five times as high) mainly resulting from QEMU.

Decoupled Taint Analysis. Dynamic taint analysis [30, 31, 32, 33, 34, 35] is a well-known technique for tracking the data flow from a source to a sink. Taint analysis is useful for runtime security policy enforcement [30, 31], malware analysis [34], and privacy leakage detection [35]. However, because of its excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [33], is six times as high), it is difficult to use it in a general computing environment. To solve this performance problem, several studies have proposed decoupled taint analysis techniques [11, 12, 36, 37]. The purpose of these techniques is to run a target process with a CPU core while performing taint analysis for a process with other idle CPU cores.

3.11 Conclusion

We presented RAIN, a practical attack investigation system with runtime efficiency and refinable granularity (from system call to instruction). By leveraging a record-and-replay technique, RAIN achieves efficiency using graph-based analysis to prune out unrelated executions, and it performs DIFT only on relevant executions to identify fine-grained causality. We demonstrated RAIN's effectiveness by applying it to an evaluation dataset to perform a precise causality analysis of sophisticated attacks.

CHAPTER 4

CROSS-HOST REFINABLE ATTACK INVESTIGATION

4.1 Introduction

Advanced attacks tend to involve multiple hosts to conceal real attackers and attack methods by using command-and-control (C&C) channels or proxy servers. For example, in the Operation Aurora [75] attack, a compromised victim's machine connected to a C&C server that resided in the stolen customers' account, and exfiltrated proprietary source code from the source code repositories. Gibler and Beddome demonstrated GitPwnd [13], an attack that takes advantage of the `git` [76] synchronization mechanism to exfiltrate victim's private data through a public `git` server. Unlike common data exfiltration attacks that only involve a victim host, GitPwnd leverages two hosts (victim's host and public `git` server) to complete the exfiltration.

Unfortunately, existing attack investigation systems, also known as provenance systems, are inadequate to figure out the true origin and impact of cross-host attacks. Many provenance analysis systems (such as [22, 16, 77]) are designed to monitor the system-call-level or instruction-level events within each host while ignoring cross-host interactions. In contrast, network provenance systems [78, 49, 79] focus on the interaction between multiple hosts, but, because they lack detailed system-level information, their analysis could result in a *dependency explosion problem* [23, 77]. To fully understand the steps and end-to-end information flow of a cross-host attack, it is necessary to collect accurate flow information from individual hosts and correctly associate them to figure out the real dependency.

Extending existing provenance systems to investigate cross-host attacks is challenging because problems of accuracy, performance, or both can be worse with multiple hosts. Although collecting coarse-grained provenance information (e.g., system-call-level informa-

tion) introduces negligible performance overhead, it cannot accurately track dependency explosion and undefined program behaviors (e.g., memory corruption) even within a single host. That is, if we associate the coarse-grained provenance information from different hosts using another vague link (e.g., network session [78, 49, 79]), the result will contain too many false dependencies. Fine-grained provenance information, (e.g., instruction-level information from dynamic information flow tracking (DIFT)), is free from such accuracy problems. However, it demands many additional computations and consumes huge memory, which will increase according to the number of hosts. More seriously, existing cross-host DIFT mechanisms piggyback metadata (i.e., *tags*) on network packets and associate them during runtime [71, 70], which is another source of huge performance degradation.

To perform efficient and accurate information flow analysis in the investigation of cross-host attacks, we propose a record-and-replay-based data flow tagging and tracking system, called RTAG. Performing cross-host information flow analysis using a record-and-replay approach introduces new challenges that cannot be easily addressed using existing solutions [77, 9, 71, 70]: that is, long analysis time and huge memory consumption. First, the communication between different hosts (e.g., through socket communication) introduces information flows that require additional information and procedure for proper analysis. Namely, the DIFT analysis requires transfer of the analysis data (i.e., tags) between the hosts in a synchronized manner. Existing record-and-replay solutions have to *serialize* the communication between hosts to transfer tags because no synchronization mechanism is implemented, leading to longer than necessary analysis time. Second, because a number of processes can run on multiple hosts under analysis, the memory requirement for DIFT instances could become tremendous, especially when multiple processes on different hosts interact with each other.

To overcome these two challenges, RTAG decouples the *tag dependency* (i.e., information flow between hosts) from the analysis with *tag overlay* and *tag switch* techniques (§4.6), and enables DIFT to be *independent* of any order imposed by the communication. This

new approach enables the DIFT analysis to happen for multiple processes on multiple hosts in *parallel* leading to a more efficient analysis. Also, RTAG reduces the memory consumption of the DIFT analysis by carefully designing the *tag map* data structure that tracks the association between tags and associated values. Evaluation results show significant improvement both in analysis time, decreased by 60%–90%, and memory costs, reduced by up to 90%, with realistic cross-host attack scenarios including GitPwnd and SQL injection.

This work makes the following contributions:

- **A tagging system that supports refinable cross-host investigation.** RTAG solves "tag dependency coupling," a key challenge in using refinable investigation systems for cross-host attack scenarios. RTAG decouples the tag dependency from the analysis which spares the error-prone orchestrating effort on replayed DIFTs and enables DIFT to be performed independently and in parallel.

- **DIFT runtime optimization.** RTAG improves the runtime performance of doing DIFT tasks at replay time in terms of both time and memory. By performing DIFT tasks in parallel, RTAG reduces the analysis time by over 60% in our experiments. By allocating an optimal tag size for DIFT based on system-call-level reachability analysis, RTAG also reduces the memory consumption of DIFT by up to 90% compared with previous DIFT engines.

The rest of this chapter is organized as follows: §4.2 describes the background of the techniques that supported RTAG's realization. §4.3, §4.4, and §4.5 present the challenges, an overview and the threat model of RTAG; §4.6 presents the design of RTAG; More specifically, §4.6.1 describes the data structure of RTAG, §4.6.3 explains how RTAG facilitates the independent DIFT; §4.6.4 describes how RTAG conducts tag switch for DIFT, and §4.6.6 presents the tag association module and how RTAG tracks the traffic of IPC. §4.7 gives implementation details and the complexity. §4.8 presents the results of evaluation. §4.9 summarizes related work, and §4.10 concludes this chapter.

4.2 Background

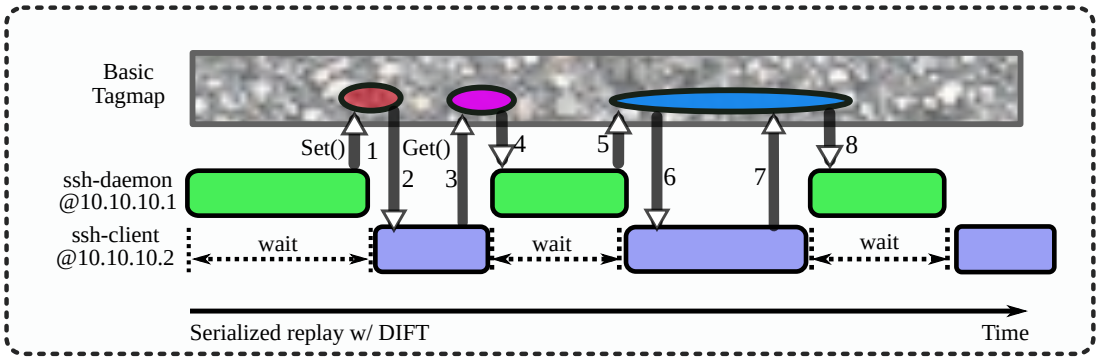
RTAG utilizes concepts from a variety of research areas. This section provides an overview of these concepts needed to understand our system.

4.2.1 Execution Logging

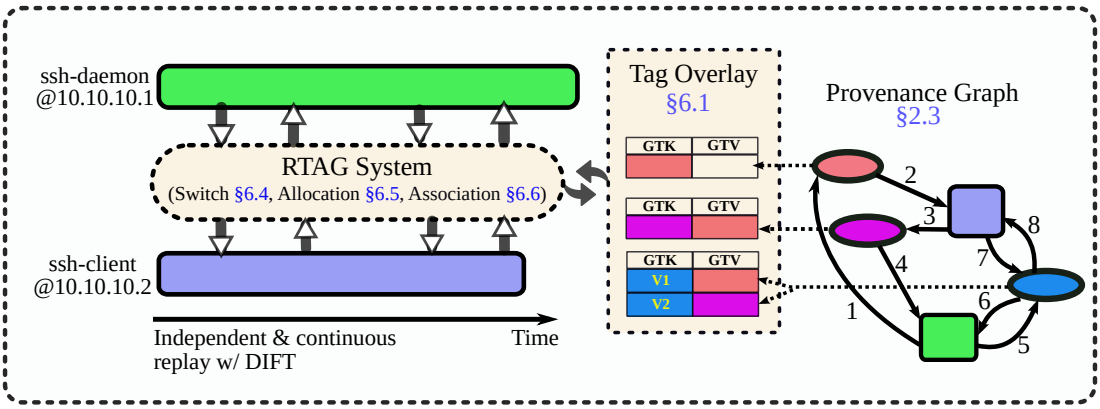
Attack investigation systems most often rely on logged information to perform their analyses. Different systems use different levels of granularity when logging information for their analyses (e.g., system-call level versus instruction level) as the cost of collecting this information changes based on the selected granularity level. A first category of systems [18, 80, 16, 22] collects information at a high-level of granularity (e.g., system-call level) and generally have low runtime overhead. However, the information collected at this level of granularity might affect the accuracy of their analyses as it does not always provide all of the execution details. A second category of systems improves accuracy by analyzing program executions at the instruction level [81, 32, 34]. These systems provide very accurate results in their analyses. However, they introduce a runtime overhead that is not suitable for production software. Finally, a third category of systems [77, 9] combines the benefits of systems from the previous two categories using record and replay. These systems perform high-level logging/analysis while recording the execution of programs and perform low-level logging/analysis in a replayed execution of the programs. More specifically, RAIN logs system call information about user-level processes using a kernel instrumentation approach. The system then analyzes instructions in a replayed execution of the processes.

4.2.2 Record and Replay

Record and replay is a technique that aims to store information about the execution of a software system (record phase) and use the stored information to re-execute the software in such a way that it follows the same execution path and also reconstructs the program



(a) Serialized DIFTs following tag dependencies



(b) RTAG independent and parallel DIFTs

Figure 4.1: Comparison of the serialized DIFTs and RTAG parallel DIFTs.

states as the original execution (replay phase). Record and replay techniques can be grouped under different categories based on the layer of the system in which they perform the record-and-replay task. Some techniques perform record and replay by instrumenting the execution of programs at the user level [40, 59, 82, 83, 84]. These techniques are efficient in their replay phase as they can directly focus on the recorded information for the specific program. However, these techniques either require program source or binary code for instrumentation or have additional space requirements when recording executions of communicating programs (especially through the file system) as the recorded information is stored multiple times. The second category of techniques performs record and replay by observing the behavior of the operating system. Techniques do so by either monitoring the operating system through a hypervisor [85, 41, 42] or emulation [43]. These techniques are efficient in storing the information about different executing programs. However, they usually need to replay every program recorded even when only one program is of interest for attack investigation. Finally, a third category of techniques uses an hybrid approach. This category records information at the operating system level and replays the execution leveraging user-level instrumentation [77, 9] (e.g., by hooking `libc` library) for multi-thread applications. More specifically, Arnold [9] and RAIN reside inside the kernel of operating system and record the non-deterministic inputs of executing programs. The replay task is achieved by combining kernel instrumentation with user-level instrumentation so that replay of a single program is possible.

4.2.3 Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT) is a technique that analyzes the information flowing within the execution of a program. This technique does so by: (1) marking with tags the "interesting" values of a program, (2) propagating tags by processing instructions, and (3) checking tags associated with values at specific points of the execution. There are several instantiations of this technique [33, 32, 34, 11, 12, 37]. These instantiations

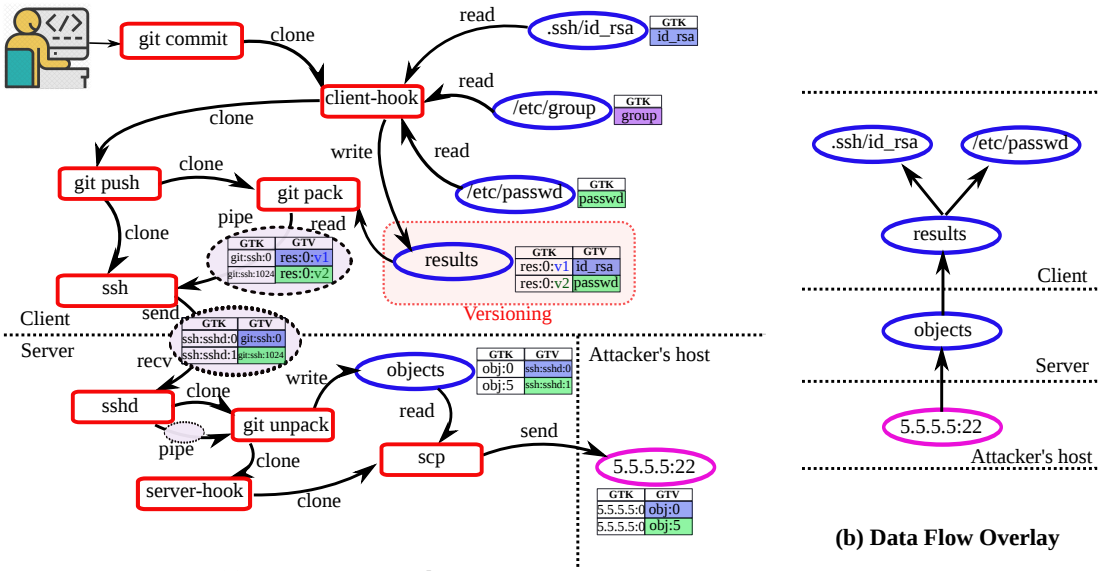
can precisely determine whether two values of the program are related to each other or not. However, because the technique needs to perform additional operations for every executed instruction, that action generally introduce an overhead which makes it unsuitable in production. Arnold [9] and RAIN make dynamic information flow tracking feasible by moving the cost of the analysis away from the runtime using a record-and-replay approach that performs DIFT only in the replayed execution. RAIN also improves the efficiency of the analysis when considering an execution that involves multiple programs. RAIN does so by: (1) maintaining a provenance graph that captures the high-level relations between programs; (2) performing reachability analysis on the provenance to discard executions that do not relate to the security task under consideration and instead pinpointing the part of the execution where the data-dependency confusion exists (i.e., memory overlaps, called *interference*); (3) performing DIFT only for interferences by replaying the execution and fast-forwarding to that part.

4.3 Motivating Example and Challenges

In section, we describe the challenges of performing refinable attack investigation across multiple hosts. We first present a motivating attack example (GitPwnd [13]) involving multiple hosts in a data exfiltration; then, we present what challenges we face with currently available methods.

4.3.1 The GitPwnd Attack

GitPwnd uses a popular versioning control tool `git` to perform malicious actions on a victim's host and sync the result to an attacker's controlled host via a `git` server. Unlike conventional data exfiltration attacks, this attack involves multiple hosts (i.e., a victim's host and the `git` server) to achieve the exfiltration. This attack evades an existing network-level intrusion detection system, as the victim's host does not have a direct interaction with any untrusted host (i.e., the attacker's host). In addition, this attack appears to be innocuous



(a) Provenance Graph & Tags

(b) Data Flow Overlay

Figure 4.2: Visualized Pruned Provenance Graph and Tags.

inside the developers' network, as `git` operations are usually assumed to be benign. We implement this attack using `gitolite` [86] at the server side and `git` at the client side.

The starting point of the attack is a malicious mirror of a popular `git` repository, which includes a hooking script that clones a command-and-control (C&C) repository for future communication. Whenever a developer (a victim host) happens to clone the malicious mirror, the `git` client will automatically clone the C&C repository as well due to the hooking script. The C&C repository includes agent and payload, whose executions will be triggered by a certain `git` operation (e.g., `git commit`) by the developer. Their execution results are saved and synced to the C&C repository. Note that the C&C repository shares the privilege of the malicious mirror repository, so it also is white-listed by the developer's host. Whenever the C&C repository receives the exploit results (stored into `objects`), it shares the results with the attacker's host (via `scp`). More specifically, this `git push` involves three processes. 1) The `git` first forks an `ssh` process, handling the ssh session with the remote host, and then 2) spawns another `git pack` process packing the related objects of the push. 3) The `pack` process uses `pipe` to transfer the packed data to the `ssh` process. The communication between the C&C repository and the attacker's host is invisible to the victim. We visualize an abbreviated pruned provenance subgraph of the attack in Figure 4.2(a). (a) is the simplified provenance graph of the GitPwnd attack involving three hosts, of which the `git` client and `git` server are monitored by RTAG. We use red rectangles to represent processes, blue ovals for file objects, and pink ovals for out-of-scope remote host; we use directed edges to represent the data flows and parent-child relations between processes. The tags with dashed circles are the IPC tags for `pipe` and `socket` communication. (b) is the result of a backward query from the attacker's host, the data flow overlay; it appears to be a tree, giving the data flow every step from the exfiltrated private key and `/etc/passwd` (excluding `/etc/group`) to the attacker's host, crossing three hosts.

We will continue to use this attack as a running example throughout the rest of this chapter.

4.3.2 Challenges

Satisfying both the accuracy and the efficiency for cross-host data flow tracking are challenging. Existing provenance systems that support cross-host accurate data flow capturing [71, 70] rely on performing DIFT at the *runtime*, which *naturally* propagates the tags from the execution of a program to another host without losing any tags and their dependencies. Unfortunately, such systems suffer from 10×–30× runtime overhead, making them impractical in production systems. Instead, to ensure both runtime efficiency and accurate data flow tracking, *refinable* systems [77, 9] record the execution of every process in the system, and *selectively* replay some of them related to the attack with DIFT instrumentation. However, existing refinable systems are subject to a tag-dependency challenge that requires the replay and DIFT of every process to be performed in the same order as the recording if a dependency exists in tags involved in different replayed processes. The enforcement of the order requires the DIFT tasks to wait for their upstream DIFTs to update the tag values that they depend on. Although the record-and-replay function can faithfully re-construct the program states at replay time, it still takes non-trivial (and error-prone) efforts to serialize and orchestrate the replays of different processes to re-establish the dependencies for tag propagation between different hosts.

The tag-dependency challenge becomes outstanding when we aim to replay processes on multiple hosts to investigate cross-host attacks. This is because the interactive two-way communication (for the purpose of network or application-level protocol) demands the replays to be paused and waiting iteratively for enforcing the same tag dependency as the recording, which further lengthens the waiting time (i.e., analysis time consumption), and increases the complexity of replay orchestration.

Let us look into one example of replay from the Gitpwnd attack [13] (detailed in §4.3.1) for the communication between the client-side `ssh` and the server-side `sshd` in Figure 4.1(a). We highlight the components of RTAG with dashed circles. (a) shows the serialized DIFT for the ssh daemon on the server and the ssh client on another host, both of which follow

the tag dependencies same as those were recorded. (b) depicts that RTAG decouples the tag dependency from the replays of processes by using the tag switch, allocation and association techniques so that each process in the offline analysis can be performed independently. At the server side, the replay of `sshd` needs to be *paused* to wait for the replay of `ssh-client` at the client side to fulfill the propagation results in the tag map for the traffic. Furthermore, this traffic will be used by `sshd` to respond to `ssh` as an `ssh` protocol response, which means the replay of `ssh` needs to be paused and wait for `sshd` as well.

This challenge is exacerbated when many parties are involved in group communication. For example, to enforce the tag dependencies for the operation of searching and downloading a file from a peer-to-peer (P2P) file sharing network (e.g., Gnutella [87]), we need to orchestrate the replays of P2P clients on each node, in which case the approach becomes infeasible particularly when we are faced with hundreds or thousands of nodes. §4.8 shows the DIFT time cost and compares it with RTAG in Table 4.1.

To systematically overcome the tag-dependency challenge, we propose RTAG that *decouples* the tag dependencies from the replays by using symbolized tags with optimal size for each *independent* DIFT. We show RTAG effectively solves the challenge while significantly speeding up DIFT tasks and reducing their memory consumption.

4.4 Overview

We propose a tagging system, RTAG, that decouples the tag dependency from the analysis (i.e., DIFT tasks), which previously was *inlined* along with the program execution or its replayed DIFT, and enables DIFT to be *independent* of any required order—allowing performing DIFT for different processes on multiple hosts in *parallel*. Such independence spares the complex enforcement of orders during the offline analysis. Note that our parallel DIFT concerns *inter*-process (or host) DIFT, which is orthogonal to the *intra*-process parallel DIFT techniques in [37, 12, 36].

RTAG maintains a *tagging overlay* on top of a conventional provenance graph, enabling

independent and accurate tag management. First, when DIFT is to be performed, RTAG uses a *tag switch* technique to interchange a *global* tag that is unique across hosts and a *local* tag that is unique for a DIFT instance. Using a local tag for each DIFT disentangles the coupling of tags shared by different DIFT tasks. After the DIFT is complete, RTAG switches the local symbol back to its original global tag. Second, to ensure no tag as well as their propagation to other tags is lost when the tag of a piece of data is updated more than once, RTAG keeps track of each change (*version*) of the data according to system-wide write operations. Each data version has its own tag(s) and each version of tag values can be correctly propagated to other pieces of data. Figure 4.1(b) depicts how RTAG facilitates the independent replay and DIFT for the cross-host `ssh` daemon and client example with the tag overlay and a set of techniques (i.e., tag switch, allocation, and association).

RTAG not only speeds up the analysis by enabling independent DIFT, but also reduces the memory consumption when DIFT is performed. We allocate local symbols of each DIFT with the *optimal* symbol size that is sufficient to represent the entropy of data involved in the memory overlap (i.e., "interference") in each DIFT (§4.6.5). For tracking the data communication across hosts, RTAG applies a *tag association* method (§4.6.6) to map the data that are sent from one host and the ones that are received at another host at byte level, which facilitates the identification of tag propagation across hosts.

4.5 Threat Model and Assumptions

In this section, we discuss our threat model and assumptions. The goal of our work is to provide a system for refinable cross-host attack investigation through DIFT. This work is under a threat model in which an adversary has a chance to gain remote access to a network of hosts, and will attempt to exfiltrate sensitive data from the hosts or to propagate misinformation (i.e., manipulate data) across the hosts. Our trusted computing base (TCB) consists of the kernel in which RTAG is running, and the storage and network infrastructure used by RTAG to analyze the information collected from the hosts under analysis. Our TCB

surface is similar to the one assumed by other studies [22, 16, 19, 77].

We make the following assumptions. First, attacks will happen only after RTAG is initiated (for collecting the information about attacks from the beginning to the end). Note that partial information about attacks can still be collected even if this assumption is not in place. Second, attacks relying on hardware trojans and side/covert channels are outside the scope of this work. Although RTAG does not yet consider these attacks, we believe a record-and-replay approach has the potential to detect similar attacks as presented in related work [56, 57]. Third, we assume that although an attacker could compromise the OS or RTAG itself, the analysis for previous executions is still reliable. That is, we assume the attacker cannot tamper with the data collected and stored from program executions of the past. This can be realized by leveraging secure logging mechanisms [49, 50] or by managing the provenance data in a remote analysis server. Finally, we assume that the attacker cannot propagate misinformation by changing the payload of network packets while they are being transferred between two hosts (i.e., there is no man-in-the-middle attack).

4.6 Tagging System

We present the design of RTAG tagging system in this section. First, we describe the design of the *tag overlay* and how it represents and tracks the data provenance in the cross-host scope §4.6.1. Second, in §4.6.2, we recall the reachability analysis from RAIN and how it is extended for the cross-host case and benefits the tag allocation. Third, we explain how RTAG decouples the tag dependencies from the replays (§4.6.3), and the tag switch technique (§4.6.4). Fourth, we explain how we optimize the local tag size in pursuit of memory cost reduction in the DIFT. Fifth, we describe how to associate tags in the cross-host communication §4.6.6. Finally, we present the investigation query interface in §4.6.7.

4.6.1 Representing Data Flow and Causality

To track the data flow between files and network flow across different hosts, we build the model of tags as an overlay graph on top of an existing provenance graph (such as RAIN). Within the overlay graph, RTAG associates globally unique tags with interesting files to track their origin and flows at *byte-level* granuality. The tags allow RTAG to trace back to the origin of a file including from a remote host and to track the impacts of a file in the forward direction even to a remote host. With this capability, RTAG extends the coverage of the refinable attack investigation [77] to multiple hosts. The provenance graph is still necessary to track the data flows: 1) from a process to a file; 2) from a process to another process; and 3) from a file to a process. An edge indicates an event between two nodes (e.g., a system call such as one that a process node reads from a file node).

In the overlay tag graph, each byte of a file corresponds to a tag *key*, which uniquely identifies this byte. Each tag key is associated with a vector of *origin* value for this key (i.e., this byte). By recursively retrieving the value of a key, one obtains all of the upstream origins starting from this byte of data in a tree shape extending to the ones at a remote host. Reversely, by recursively retrieving the tag key of a value, the analyst is able to find all the impacts in a tree shape including the ones at a remote host (see Figure 4.2(b) as an example).

As we log the system-wide executions, RTAG needs to uniquely identify each byte of data in the file system on each host as a "*global tag*." For this requirement, RTAG uses a physical hardware address (i.e., `mac` address) to identify a host, identifiers such as `inode, dev, crtime` to identify a file, and an offset value to indicate the byte-level offset in the file. For example, the physical hardware address (i.e., `mac` address) is 48 bits long. The `inode, dev, crtime` are 64 bits, 32 bits, and 32 bits consecutively. The offset is 32-bits long, which supports a file as large as 4GB. Thus, in total, the size of a global tag can be 208 bits.

4.6.2 Cross-host Reachability Analysis

RTAG follows the design of reachability analysis in RAIN, and extends it to cope with the cross-host scenarios. Given a starting point(s), RTAG prunes the original system-wide provenance graph to extract a subgraph related to the designated attack investigation that contains the causal relations between processes and file/network flow. RTAG relies on the coarse-level data flows in this subgraph to maintain the tag overlay while performing tag switch and optimal allocation. The reachability analysis first follows the time-based data flow to understand the potential processes involved in the attack. Next, it captures the memory overlap of file or network inputs/outputs inside each process and labels them as "interference," to be resolved by DIFT. With accurate interference information, the replay and DIFT are fast forwarded to the beginning of the interference (e.g., a `read` syscall) and early terminated at the end (e.g., a `write` syscall).

For the network communication crossing different hosts, RTAG links the data flow from one host to another by identifying and monitoring the socket session. As we present in §4.6.6, RTAG tracks the session by matching the IP and port pairing between two hosts. RTAG further tracks the data transfer at byte level via socket communication for both TCP and UDP protocols, which enables the extension of tag propagation across hosts. Unlike the runtime DIFT system, RTAG has the comprehensive knowledge of source and sink from the recorded file/network IO system-call trace, thus is able to allocate an optimal size of tag for each individual DIFT task. We show in §4.6.5 that this optimization significantly reduces the memory consumption of DIFT tasks. In addition, to avoid losing any intermediate tag updates to the same resource performed by different processes, RTAG particularly monitors the "overwrite" operations to the same offset of a file and tracks this versioning info, so it accurately knows which version of the tag should be used in the propagation.

4.6.3 Decoupling Tag Dependency

As a refinable provenance system, RTAG aims to perform DIFT at the offline replay time without adding high overhead to the runtime of the program. The replay reconstructs the same program status as the recording time by enforcing the recorded non-determinism to the replay of process execution. The non-determinism includes the file, network, and IPC inputs which are saved and maintained with a B-tree [9]. Such enforcement enables the program to be faithfully replay-able at process level.

To extend this approach to capture the end-to-end data flow across multiple hosts, we need to figure out how to coordinate replay programs on different hosts to track tag dependencies between them. One possible method is decoupling tag dependencies from each replay of the process, so it can be performed independently with no dependency on other replays. We achieve the decoupling by using *local* (i.e., *symbolized*) tags for each DIFT. Such symbolization needs to distinguish the change of a tag before and after the `write` operation on it, and synchronize the change to other related tags as well. In other words, RTAG needs to track the dynamic change of origin(s) of each tag after each IO operation (i.e., multiple *versions* of the tag are tracked).

Let us illustrate with the data exfiltration in the Gitpwnd attack example in Figure 4.2(a). The `client-hook` daemon keeps reading data from different files (e.g., `/etc/passwd`, `id_rsa`) and saves them into a `results` file which is recycled over a period of time. Meanwhile the `git pack` application copies from the `results` file whenever the victim does `git commit` operation, and shares data with `ssh` via the `pipe` IPC, which will be shipped off the host. To correctly differentiate the two data flows, `id_rsa`→`results`→`pipe` and `/etc/passwd`→`results`→`pipe`, RTAG needs to maintain two versions of the tags for `results`. The DIFT on `client-hook` stores the origin of `results.v1` to be `id_rsa`, and the origin of `results.v2` to be `/etc/passwd` (circled with red dash line), while the DIFT on `git pack` is able to discriminate the source of the IPC traffic `git:ssh` at offset 0 from `results.v1` and further from `id_rsa`, and the source of the IPC traffic at offset 1024 from

`results.v2` and further from `/etc/passwd`. Most importantly, now the `client-hook` and `git pack` DIFT tasks can be performed independently without losing intermediate tag values because of the overwriting on `results`.

To facilitate the versioning, we append a 32-bit "version" field to indicate the version of the data in the file with regards to the file IO operation. According to the sequential system-call trace, the version is incremented at every event in which there is a write operation against this certain byte (e.g., `write()`, `writev()`). In the case of memory mapped file operation (e.g., `mmap()`), the version is incremented at the `mmap()` if the `prot` argument is set to be `PROT_WRITE`. The version field is only used when this tag is included in the data interference determined by the reachability analysis. We assign 32 bits for this field that can pinpoint a file IO syscall in around 500 days based on our desktop experiment.

4.6.4 Switching Global and Local Tags

The entropy of the global tag defined in §4.6.1 is sufficient enough to identify a byte of a file at a certain version across multiple hosts. However, using the global tag for each DIFT task is a waste of memory because each DIFT task of RTAG only covers a process group such that a *local* tag ensuring process-group-level uniqueness is enough. Thus, for each DIFT task, we use a different tag size based on the entropy of its source symbols. RTAG switches the tags from *global* to *local* before doing DIFT, and switches them back when the DIFT is done. The tag for DIFT is local because it only needs to uniquely identify every byte of the source in the current in-process DIFT, rather than identify a single byte of data across multiple hosts.

Further, the number of sources in each DIFT depends on the reachability analysis result, which is usually largely reduced by data pruning. In other words, the local tag size depends on the *interference* situation. Therefore, the entropy for the local tag is much lower than the global tag. For example, if the program reads only 10 bytes from a file marked as a source in DIFT, in fact as low as four bits are sufficient to represent each of these bytes.

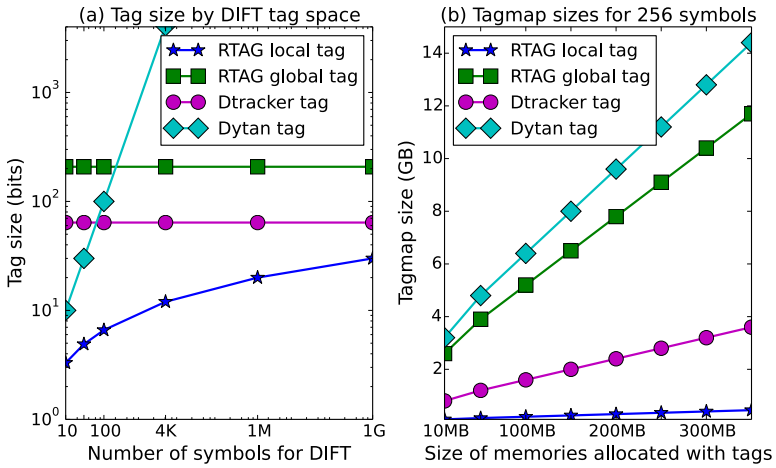


Figure 4.3: Memory cost for tags in DIFT.

Compared against the global tag size (i.e., 208 bits §4.6.1), the switch brings 52× reduction in tag size (in practice, the reduction can be as large as 26× capped by the compiler-enforced byte-level granuality, which we discuss in detail in §4.7). Moreover, the tag size affects not only the symbols for the source and sink, but also all the intermediate memory locations and registers because the tags are copied, unioned, or updated along with the execution of each instruction according to the propagation policy of DIFT. Therefore, the tag size literally affects the memory cost of the whole tag map and tag switching significantly reduces the overall memory cost of DIFT.

4.6.5 Optimal Local Tag Allocation

The runtime cost of DIFT is high, both in time and storage. DIFT usually takes 10×–30× longer than the original execution because its instrumentation adds additional tag update operations to each executed instruction. Recent studies [11, 12] alleviate this issue by decoupling the instrumentation efforts from the runtime of the program. However, the storage footprint of *tag map*, the data structure used by DIFT to maintain the tag propagation status, can still be very high particularly when there are *multiple* (or *many*) sources.

The cost of tag map in DIFT depends on its supported type of tags and purpose. DIFT engines such as Taintcheck [30], Taintgrind [88], and ShadowReplica [11] use a basic binary

tag model for DIFT, which assigns a boolean "tainted" or "not tainted" for each source of DIFT. It is able to tell whether the tainted data is propagated to the sink, which can be used to alarm sensitive data leakage or control-flow hijacking. However, this model is not flexible enough for the goal of RTAG, where the data dependency confusion it aims to resolve involves *multiple* sources.

Dytan [32] and DataTracker [45] provide a customizable model for the data sources and sinks. It allows the allocation of multiple tags to each addressable byte of data at the source or sink. The tag model used by such systems is flexible, but the tag map used to maintain the status of the taint propagation is "over-flexible" thus huge, which inhibits the deployment of such a system in many resource restrained cases. As these systems assume to be running at the runtime of a program, where no knowledge of the data at the source or sink is known prior, they usually assign a fixed size for each tag such that they are confident it is safely big enough. For example, DataTracker [45] uses 32 bits to identify an inbound file, and another 32 bits to identify the offset of the data (totally 64 bits). The size is sufficient for identifying every byte in a normal desktop. Dytan [32] represents whether one source is tainted or not as one bit and stores all the bits in a bit vector as the tag. Thus the size of each tag is *linear* to the number of sources, which can be huge in the case of a high number of sources. Note that the tag map not only stores the tags for the source and sink, but all the intermediate memory locations and registers as well. Since most implementations of DIFT maintain the tag map in memory to pursue faster instrumentation, such high use of memory has a possibility to cause the DIFT to crash before it is complete. This problem is elevated when the scope of investigation extends to multiple hosts since the workload of DIFT increases in proportion.

In contrast to the previous works that perform DIFT at the program runtime, RTAG is a record-replay based system in which the knowledge of data source and sink is *known* to us when we perform DIFT at replay time. In other words, we know which (bytes of) data need to be involved in the DIFT. Thus, we can adjust the tag size based on the entropy of the data

dependency confusion, rather than use a fixed-size tag. Figure 4.3 compares the memory cost for tag map in different DIFT engines. The left (a) shows the size of each tag given different numbers of symbols used in DIFT. The right (b) depicts the tagmap sizes based on different sizes of memories being allocated with tags when 256 symbols are used in the DIFT. RTAG local, global, DataTracker, and Dytan tags are compared. (a) shows that the local tag of RTAG grows in logarithm while others are either linear or constant; (b) presents the total tag map size under different sizes of memories that are tainted (i.e., allocated with tags) where the memory cost introduced by RTAG is the lowest (by significant difference). Before DIFT, RTAG computes the optimal local tag needed to mark the source and substitute the global tag for the local one when a source is loaded to the memory space of the process (e.g., via `read()` syscall). While performing DIFT, RTAG allocates the tags for intermediate locations *lazily* when a memory location or register becomes tainted with some tag. When the propagation arrives at a sink (e.g., via a `write()` syscall), RTAG replaces the local tag with the original global one, and updates the tag value of the sink. We observe significant memory cost reduction by applying this optimal tag allocation method (see §4.8.2).

4.6.6 Tag Association

In order to track the data flow between different hosts, we additionally hook the socket handling of the operating kernel to enable the cross-host tagging. Prior studies adopt an "out-of-band" method to track the data flow communication (e.g., [67, 71]). Though this method is more straightforward when identifying and managing the tags across hosts, it requires additional bookkeeping that incurs both complexity and overhead to the hosts. In contrast, we propose an "in-band" method to track the data flow among hosts, which particularly fits the system-level reachability analysis as well as the DIFT.

We design the cross-host tagging method based on the characteristics of the socket protocols. Our current tagging scheme supports the two major types of protocols (i.e., TCP [89] and UDP [90]). For TCP, as the data stream delivery is guaranteed between the

two hosts, we rely on the order of bytes in the TCP session between source and destination to identify the data flow at byte level, which can be uniquely identified using a pair of IP addresses and port numbers. Such tracking silently links the outbound traffic from the source host with the inbound traffic at the destination host, which does not incur additional traffic. Note that although TCP regulates the data stream order, the sender or receiver may run different numbers of system calls in sending and receiving the data. For example, the sender may perform five `writev()` system calls to send 10,000 bytes of data (2,000 bytes each call), while the receiver may conduct 10 `read()` calls (1,000 bytes each call) to retrieve the complete data. This is why counting sent or received bytes is necessary, instead of counting the number of system calls.

In the case of UDP, since the data delivery is not guaranteed, some UDP packets could be lost during transmission. So we cannot rely on the order of transferred bytes because the destination host has no knowledge of which data are supposed to arrive and which have been lost. To support UDP, we embed a small "cross-host" tag at each `send` related system call by the source host, and parse the tag at `receive` related system calls by the destination host. The tag is inserted into the beginning of the datagram as a part of the user datagram before the checksum is calculated. If the datagram is transferred successfully, RTAG knows a certain length of data goes from the source to the destination. If the destination host finds the received datagram is broken, or totally lost, it will discard this datagram, hence RTAG is also aware of the loss and erases this inbound data from the reachability analysis and DIFT. As we will show in §4.8, the communication cost for TCP case is 0, while the cost for UDP is also marginal in the benchmark measurement.

The cross-host tag represents the byte-level data in the socket communication between two processes across hosts. Each tag key represents the data traffic in one socket session using the source and destination process credentials, plus the offset that indicates the data at byte level. For the uniqueness of session, we use the process identifier (`pid`) and the process creation time (`start_time` in the `task` structure) to identify each process. The tag values

represent the origin of the tag key, which is determined by the DIFT and updated to the global tag map. The cross-host tags are also switched away before DIFT is performed and restored afterward. For the hosts on which RTAG does not run, we treat them as a black box, and identify them using the IP address and port number. The IP and port are retrieved from the `socket` structure inside the kernel.

Handling IPC. RTAG tracks the data transfer of IPC communication between two processes as well. For the IPC that uses system call as a controlling interface (e.g., `pipe`, and System V IPC: `message queues`, `semaphores`), RTAG hooks these system calls to track the data being transferred. When a process uses `pipe` to send data to the child process, RTAG monitors the `read` and `write` system calls to track the transferred data in bytes. During reachability analysis, we create tag keys to label every byte sent from the parent to the child. The tag values are fulfilled by DIFT. For example, in Figure 4.2, although the `git pack` and `ssh` processes have IPC dependency, RTAG is able to perform the replay and DIFT independently on them since RTAG caches the inbound data reads from the pipe and feeds them back during the replay. Also, by tracking the `inode` associated with the file descriptors (rather than tracking `pipe, dup(2)` and child inheritance relationships), we identify the data transmitted via the pipe at byte level and the processes at its two ends. RTAG *implicitly* tracks the IPC based on shared memory. Instead of trapping the replay of a process for each read from a shared memory, RTAG replays the processes having shared memory as a *group* as RAIN and Arnold [9] do, so that the tag propagation of this shared memory is performed within the process' memory locations. No separate tag allocation is needed for these processes.

4.6.7 Query Results

The query result will be returned after all the tag values of the interfering data are updated. The result represents the data causalities of involved objects in a tree structure. For example, in Figure 4.2, a backward query on the attacker's controlled host `5.5.5.5:22` will return

the tree-shape data flow overlay depicted in Figure 4.2(b), consisting of all the segments of the flow from the key to all of its upstream origins. Also, a forward query returns every segment of the data flow from the queried tag key to all of its impact(s). It relies on a *reversed* map where the tag key and value are swapped to locate the downstream impact from a file. For example, a forward query on the private key `id_rsa` on the client side returns a flow: `id_rsa`→`results.v1`→`objects`→`5.5.5.5:22`. A point-to-point query gives the detailed data flow between two nodes in the provenance graph by performing a forward and backward query on these two nodes, then computing the intersection of the two resulting trees.

4.7 Implementation

The implementation of RTAG is based on a single-host refinable information flow tracking system RAIN, with extended development of the tagging system. Specifically, our implementation adds 830 lines of C code to the Linux kernel for the tag association module, 2,500 lines of C++ code to the DIFT engine for the tag switch mechanism, 1,100 lines of C++ code for the maintenance of tags, 900 lines of C++ code for the query handler, and 500 lines of Python code for the reachability analysis for tag allocation. Currently, RTAG runs on both the 32-bit and 64-bit Ubuntu 12.04 LTS. Accordingly, our DIFT engine supports both x86 and x86_64 architectures, which is based on libdft [33] and its extended x86_64 version from [91]. We use a graph database Neo4j [61] for storing and analyzing coarse-level provenance graphs, and a relational database PostgreSQL [92] for global tags with multiple indexing on host (i.e., MAC address) and file credentials (i.e., `inode`, `dev`, `crtime`). Particularly, we supplement the tag data structure §4.6.4 and how we track socket session §4.6.6 with implementation details in the following.

Tag Data Structure. In the current implementation, RTAG maintains local tags for individual bytes. RTAG uses C++'s `vector` as the multi-tag container for one memory location or register and uses sorting and binary search in the case of `insert` operation.

`vector` has storage efficiency, although its insertion overhead is higher than that of the `set` data structure, which was used by DataTracker [45]. We make this choice based on x86 instruction statistics [93] that show the most popularly used instructions are `mov`, `push`, and `pop` of which the propagation policy copies the tag(s), while instructions that involve insertion, such as `add` and `and`, are much less frequent. Our evaluation affirms this choice that the time overhead for single DIFT is similar between RTAG and previous work [45].

Tracking Socket Session. The implementation of tracking the socket communication session refers to the `socket` structure inside the kernel for IP and port of the host and the peer. If the type of socket is `SOCK_STREAM` (i.e., TCP), we use a counter counting the total number of bytes sent or received by tracking the return value of `send` or `write` system calls. If the type is `SOCK_DGRAM` (i.e., `UDP`), our implementation embeds a four-byte incrementing sequence number within the same peer IP and port number at the beginning of the payload buffer inside an in-kernel function `sendmsg` rather than the system call functions such as `send` and `recv` to avoid affecting the interface to the user program as well as the checksum computation. At the receiver side, we strip the sequence number in the `recvmsg` after the checksum verification and present the original payload to the program. As shown in §4.8.2, the hooking at this level incurs almost no overhead in either bandwidth or socket handling time. It also avoids the complicated fragmentation procedure at the lower level.

4.8 Evaluation

Our evaluation addresses the following questions:

- How well does RTAG handle the data flow queries (forward, backward, and point-to-point) for cross-host attack investigations? (§4.8.1)
- How well does RTAG improve the efficiency of DIFT-based analysis in terms of time and memory consumption? (§4.8.2)
- How much overhead does RTAG cause to system runtime including the network

bandwidth? (§4.8.2, §4.8.2) What is the storage footprint of running RTAG? (§4.8.2)

Settings. We run RTAG based on the Ubuntu 12.04 64-bit LTS with 4-core Intel Xeon CPU, 4GB RAM and 1TB SSD hard drive on a virtual machine using `KVM` [94] for the target hosts where system-wide executions are recorded. On the analysis host, we use a machine with 8-core Intel Xeon CPU W3565, 192 GB RAM, and 2TB SSD hard drive installed with Ubuntu 12.04 64-bits for handling the query and performing DIFT tasks in parallel. We use `NFS` [95] to share the log data between the target and the analysis host.

4.8.1 Security Applications

Table 4.1 summarizes the statistics in every stage of processing a query for an attack investigation: the original provenance graph covering all the hosts, the pruned graph where the unrelated causalities are filtered out by the reachability analysis, and the data flow overlay where the tags store the origins of each byte of data involved in the query. **Prov Graph** are the original graph containing the system-wide executions of every process. **Pruned Graph** are the subgraph where nodes and edges that are unrelated to the attack are pruned out; **DF Overlay** are results from the RTAG tagging system; **Tags** gives the number of generated tag entries; **C-Tags** gives the number of tags of which the key and value(s) are **C**ross-host (i.e., from different hosts); **Accuracy** shows the percentage of how many data flows are matched with the ground truth. Table 4.2 also summarizes how long each of the queries took and their memory consumption. **Tasks** stands for the number of processes that are replayed with DIFT; **Memory** gives the sum of virtual memory cost for each task; **Time** gives the time duration RTAG spends to perform the DIFT tasks in parallel; **TReduc%** shows the reduction rate from the time of performing the same DIFT tasks serially.

GitPwnd

We first present how RTAG handles the queries on the Gitpwnd example (described in §4.3.1). To handle a query, we replay the involved processes independently based on reachability

Table 4.1: Statistics in terms of the effectiveness and performance of cross-host attack investigation.

Items		Prov Graph		Pruned Graph		DF Overlay		Accuracy
Attack	**Query**	**Node**	**Edge**	**Node**	**Edge**	**Tags**	**C-Tags**	
GitPwnd	FW: /etc/passwd	8.3K	109K	39	557	**28,960**	**10,700**	**100%**
	BW: attacker host			55	1,661	**32,660**	**18,032**	**100%**
	PP: results - objects			22	418	**23,193**	**7,317**	**100%**
SQLi-1	FW: exploit html	5.3K	89K	33	711	**6,799**	**882**	**100%**
	BW: payroll record			29	683	**8,257**	**882**	**100%**
	PP: html - db file			27	490	**3,197**	**882**	**100%**
SQLi-2	FW: db file	5.2K	87K	80	2,251	**510,466**	**420,121**	**100%**
	BW: dump file			72	1,997	**530,004**	**420,121**	**100%**
CSRF	FW: exploit html	2.8K	34K	89	2,379	**9,224**	**1,766**	**100%**
	BW: salary record			97	2,270	**7,700**	**1,766**	**100%**
XSS	FW: exploit html	2.9K	24K	71	1,145	**432,845**	**420,755**	**100%**
	BW: attacker host			63	863	**435,716**	**420,700**	**100%**
	PP: html - a-host			55	782	**421,106**	**420,700**	**100%**
P2P	BW: mp4@12th node	13K	730K	74	240K	**759,302**	**630,228**	**100%**
	FW: mp4@1st node			182	490K	**3,088,102**	**2,532,920**	**100%**

analysis results while performing DIFT on the interfering parts. We run RTAG on both client and server hosts involved in this attack, while treating the attacker-controlled host as a black box. We perform three queries: a forward query asking for where the leaked `/etc/passwd` goes to, a backward query inquiring the sources of data flow that reaches the attacker's controlled host, and a point-to-point query aiming to particular data flow paths between the `results` file on the client side and the `objects` file on the server side. In Table 4.1, we show the statistics of using RTAG in every step. Particularly, we show the number of tags RTAG creates at the tag overlay. In the forward query, RTAG generates 28,960 tag entries totally, 10,700 of which are cross-host ones meaning the tag key and value are from different hosts. We compare the query result with ground truth of the attack and RTAG achieves 100% accuracy in every query. We also evaluate the performance improvement for DIFT, summarized in Table 4.2. In general, thanks to the parallelizing of DIFT tasks, RTAG reduces the time cost by more than 70% in most cases.

Table 4.2: DIFT performance using RTAG.

Items		DIFT Perf			
Attack	**Query**	**Tasks**	**Mem(MB)**	**Time(s)**	**TReduc%**
GitPwnd	FW	10	497	95	**87%**
	BW	27	912	113	**86%**
	PP	8	322	79	**72%**
SQLi-1	FW	14	2,513	342	**70%**
	BW	11	2,336	339	**64%**
	PP	9	1,997	309	**76%**
SQLi-2	FW	41	7,655	695	**83%**
	BW	39	6,804	677	**82%**
CSRF	FW	33	6,537	499	**78%**
	BW	49	7,122	504	**84%**
XSS	FW	26	4,850	687	**77%**
	BW	28	5,391	705	**77%**
	PP	19	4,107	677	**72%**
P2P	BW	12	6,371	201	**92%**
	FW	12	9,855	236	**91%**

Web-based Attacks

We also use a set of web-based attacks to evaluate the effectiveness of RTAG in tracking the data flow between the server (e.g., a web server `Apache`), and the client (e.g., a browser `Firefox`). The web app facilitates the checking and updating of employees' personal financial information. The employees typically manage their bank account number and routing number via the web app. The attacks include two SQL injections, one cross-site request forgery (CSRF), and one cross-site scripting (XSS). We set up RTAG on both server and client. We run an `Apache` server with `SQLite` as its database. At the client, we load exploit pages with either a data transfer tool `Curl` or the `Firefox` browser. For each attack, we perform three types of queries and compare the query results with the ground truth.

SQL injections. The exploit takes advantage of a vulnerability at the server's SQL parsing filter to execute illegal query statements that steal or tamper the server database. The first attack (**SQLi-1**) injects an entry of user profile to the database. The added profile is further used by another financial program to generate payroll records. The analyst performs a

forward query from the loaded `html` file with the exploit, and RTAG returns the data flows from the file at the client to the data in the payroll records. The second attack (**SQLi-2**) steals data entries in the database from the user and exploits a vulnerability in `Firefox` to dump the entries to a file. With a backward query from the dump file at the user side, RTAG pinpoints the segments of the database file that has been exfiltrated.

Cross-site request forgery. The exploit uses a vulnerability of the server that miscalculates the CSRF challenge response to submit a form impersonating the user. The form updates the profile contents (e.g., account number), and later the tampered profile is accessed by several other programs that process the user's payroll information. RTAG helps determine the data flow between the user's loaded file and one of the payroll record that is considered to have been tampered.

Cross-site scripting. The reflection-based cross-scripting relies on dependency of an html element to user input to append a script that reads the sensitive data from the `DOM` tree of a page, packs some of the data, and sends an email to the attacker's external host. After the investigation determines the attacker's host to be malicious, it makes a backward query from that host and finds the data exfiltration from the user's loaded page, as well as from a certain offset of the database storage file at the server. Notably, the resulting overlay shows the route of some tags tracing back first to the server side (i.e., `Apache`), then further back to the client side browser and the exploit html file, which recovers the *reflection* nature of the attack.

Attacks Involving Memory Corruptions

To evaluate RTAG for the cases when the attacker exploits memory corruptions, we additionally modified the GitPwnd attack §4.3.1 by compiling the `ssh` daemon with earlier versions containing memory-based vulnerabilities: one integer overflow based on CVE-2001-0144 and one buffer overflow based on CVE-2002-0640. For the integer overflow, we patched the `ssh` client side code to exploit the vulnerability [24] and remotely executed `scp` command

at the server to copy files to the attacker's controlled host. For the buffer overflow, we crafted a malicious response for the OpenSSH (v3.0) challenge-response mechanism and remotely executed commands [25]. We note that memory-corruption-based attacks usually involve undefined behavior of the program that violates the assumption of many previous investigation systems using source or binary semantics (e.g., [23, 12, 11]). However, RTAG successfully reconstructs the program state of the overflow for the DIFT to recover the fine-grained data flow.

File Spreading in Peer-to-Peer Network

We also run RTAG to track the data flows in a malicious-file-spreading incident on top of a P2P network, which is regarded an increasing threat in the decentralized file sharing, according to a report by BitSight Insight [96]. This allows us to demonstrate RTAG's ability to handle a complex cross-host data-flow analysis involving multiple parties, which is infeasible with existing approaches. We use Gtk-Gnutella [87](v1.1.13) to set up a P2P network in a local network of 12 nodes with RTAG running on them. We perform two operations. First, we have two nodes online; one node shares a malicious audio `mp4` file, and another node searches for the file, discovers it and downloads it. Later, we shutdown the first node and let a third node download the file from the second node. We performed this type of single-hop relay iteratively until five nodes have this file. Second, we use these five nodes as "seeds" and let the remaining nodes search, discover, and download the file. During this process, we intentionally shutdown parts of the nodes to introduce "resume" procedures. Finally, we perform a backward query from the audio file at the last node to search for the origin of the file, and a forward query from the first node to uncover how the file spread across the network with fine-grained-level data flows. RTAG returns the results with 100% accuracy. Particularly, the result also shows the data flow between each pair of nodes for each iteration of the file sharing procedure. The statistics of this experiment are summarized in Table 4.1.

4.8.2 Performance

DIFT Runtime Performance

We compare the memory consumption and execution time of RTAG with previous DIFT systems. For the memory efficiency, we evaluated two state-of-the-art DIFT engines that provide multi-color symbols, Dytan [32] and DataTracker [45]. Table 4.3 shows the peak memory consumption of the tag map for various DIFT tasks we used in evaluating the security application in §4.8.1. **#Symbols** denotes the number of symbols used in performing the DIFT task; **NA** means the DIFT is not complete so the peak memory cost is not available. The peak memory consumption is useful as it indicates the required resource for a certain type of DIFT. Notably, all the tag sizes for representing the DIFT symbols determined by reachability analysis are within three bytes (i.e., up to 16,777,216 symbols), with a majority being two bytes (i.e., up to 65,536 symbols). This means the data pruning and reachability analysis effectively narrow down the scope of the DIFT symbols and pinpoint the exact bytes of data that causes the data confusion for DIFT to resolve. The savings from the tag map consumption of RTAG is between 70% and 95%. The effect of improvement on the general memory consumption varies across different programs in terms of their own memory usage.

In our experiments, DIFT reduced total memory usage 10% to 50% when compared with DataTracker [45], and by 30% to 90% compared with Dytan [32]. Since these DIFT systems are designed with the scope of one host, in order for proper comparison against previous DIFT systems, we only measured the cases where all the tags are within one host. Note that this approach only compares DIFT runtime performance side by side, but does not indicate or suggest that RTAG can only handle single-host cases. For evaluating the time efficiency in performing DIFT tasks, we assign the same DIFT tasks to RTAG as well as to the DIFT engine used by RAIN. Since RAIN does not support cross-host investigation, we use RAIN to run the DIFT tasks, sequentially simulating the time consumption it needs to

Table 4.3: DIFT Tag Map Overhead in Practice.

Programs	#Symbols	Peak TagMap Cost (MB)			Reduc%
		DataTracker	Dytan	RTAG	
git-core	247	12	19	**4.8**	**60 / 74**
ssh	16,983	5.9	630	**2.6**	**55 / 99**
cli-hook	1,983	17	140	**8.0**	**53 / 94**
Curl	56,010	4.8	1,050	**2.3**	**52 / 99**
Firefox	4,091,773	155	NA	**67.5**	**56 / NA**
Apache	2,128,700	133	NA	**41.7**	**68 / NA**

serialize the network interaction and orchestrating the replays. We observe that the parallel DIFT of RTAG takes 60%–90% less time than RAIN (Table 4.2).

Discussion. For the memory consumption, we find the taint propagation is mainly composed of copy operations such that the tag map is just updated with another value. Combination operation for merging the tags of two locations is not frequent. Hence, though bit-vector (used in [32]) ensures a constant length of tag for each location even after combination, the benefit is not obvious. On the contrary, its fixed size is linear to the number of symbols, which causes out-of-memory crash when there are many symbols to tag or (and) the many memory locations are propagated during the execution. Using `set` eases the implementation complexity as it natively supports the combination operation with a good performance. However, it incurs higher metadata cost (on x86 Linux, storing every 4-byte data in the `set` incurs over 14 bytes). For the time consumption savings in RTAG, the total time consumption depends on the longest DIFT task (e.g., `Firefox` session). We are looking into integrating in-process parallel DIFT techniques to RTAG that could further bring down the time consumption.

Runtime Overhead

We measure the runtime overhead of RTAG using two sets of benchmarks: the SPEC CPU2006 benchmark for CPU-bound use cases and the IO-intensive benchmarks for IO bound cases. The measurements are performed on two systems, one without RTAG and one with RTAG enabled. The result of SPEC benchmark is given in Figure 4.4 with RAIN as

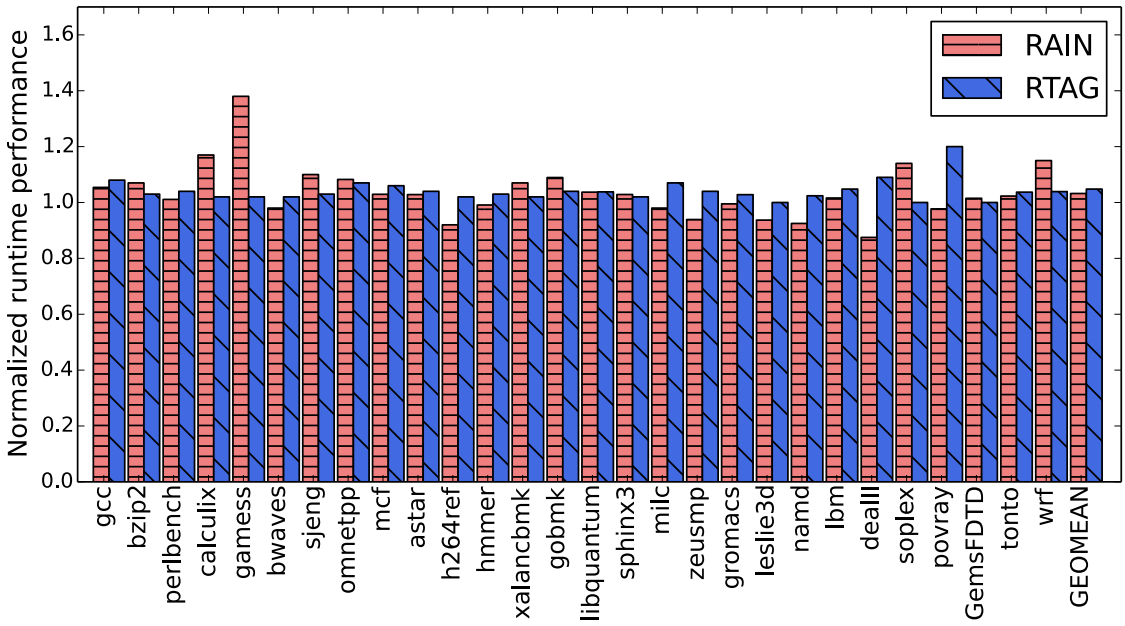


Figure 4.4: Comparison of normalized runtime performance between RAIN and RTAG with CPU bound benchmark SPEC CPU2006. "GEOMEAN" gives the geometric mean of the performance numbers.

reference. The geometric mean of the runtime overhead is 4.84%, which shows RTAG has similar low runtime overhead to previous refinable systems. We also measure the runtime overhead using IO-intensive applications to test the performance in IO bound cases. The benchmark is composed of four scenarios: using `scp` to upload a 500MB archive file, using `wget` downloading a 2GB `mov` movie file, compiling LLVM 3.8, and using `Apache` to serve an `http` service for file downloading. The result of IO-intensive applications is shown in Figure 4.5. The overhead of all the items is at most 50%. We reason that the cause of the higher overhead during file downloading and compiling is because network and file inputs are cached during the recording time.

Network Performance Impact

We use iperf3 [97] to test the bandwidth impact of applying RTAG to typical network protocol settings. For TCP, we measure the bandwidth both with and without having RTAG running at different window sizes. For UDP, we set the buffer size to be similar with real

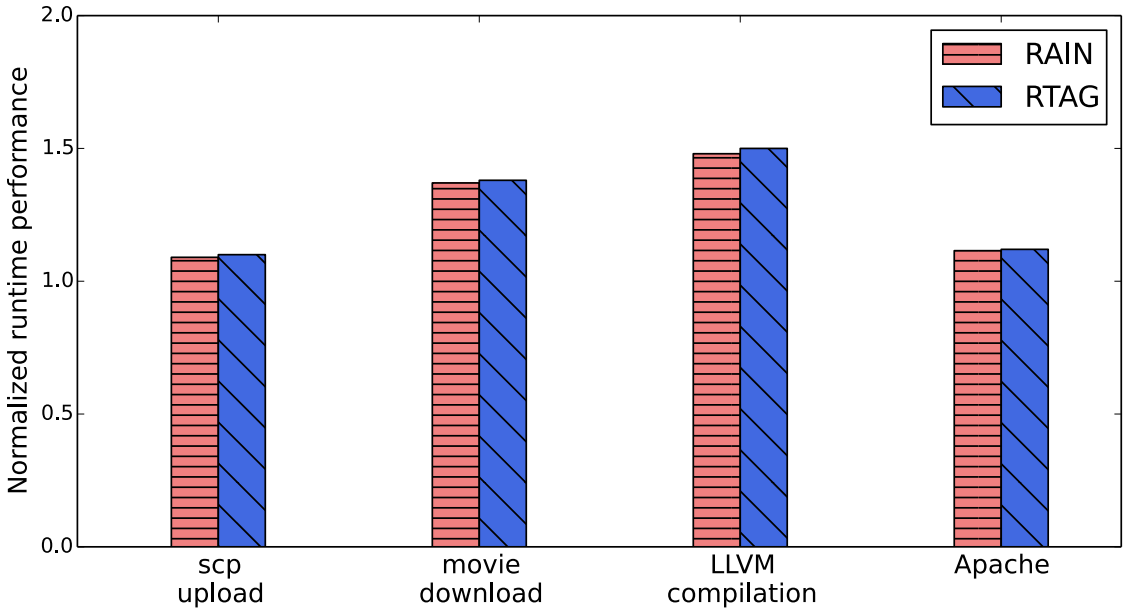


Figure 4.5: Comparison of normalized runtime performance between RAIN and RTAG with IO bound benchmarks.

Table 4.4: Bandwidth impact of RTAG.

Protocol	**Setting**	**Bandwidth%**	**RTT%**
TCP	Window: 128KB	0%	+0.03%
	256KB	0%	+0.01%
	512KB	0%	+0.012%
UDP	Buffer: 512B	-0.8%	+0.02%
	8KB	-0.05%	+0.01%
	128KB	-0.01%	+0.012%

applications such as DNS (512B), RTP (128KB). We also measure the performance impact in the term of the end-to-end round-trip-time (RTT) for one datagram to be delivered to the server and echoed back to the client. Both impacts are negligible. The results are summarized in Table 4.4. The bandwidth and round-trip-time (**RTT**) are measured with iperf3 benchmark using different settings for TCP and UDP protocols.

Storage Footprint

As a refinable system, RTAG has the storage overhead for the non-deterministic logs that are used for faithful replay of the recorded system-wide process executions. This ensures the

completeness of retroactive analysis particularly for the advanced low and slow attacks. The storage footprint varies according to the workload on each host and is comparable with the upstream system RAIN. Note that only the input data are stored as non-determinism, thus in the multi-host case, the traffic from a sender to a receiver are only stored at the receiver side, avoiding duplicated storage usage. In the use of RTAG, we observe around 2.5GB–4GB storage overhead per day for a desktop used by a lab student (e.g., programming, web browsing); and around 1.5GB storage overhead per day for a server hosting gitolite used internally by five lab students for version controlling on course projects.

4.9 Related Work

Dynamic Information Flow Tracking. Dynamic taint analysis [32, 33, 35, 98, 30] is a well-known technique for tracking information flow instruction by instruction at the runtime of a program without relying on the semantic of a program source or binary. DIFT is useful for policy enforcement [30], malware analysis [34], and detecting privacy leaks [35, 98]. To support intra-process tainting, Dytan [32] provides a customizable framework for multi-color tags. DataTracker adapts standard taint tracking to provide adequate taint marks for provenance tracking. However, taint-tracking suffers from excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [33] is six times as high as native execution), which makes it difficult to use in a runtime environment. To solve this problem, several approaches have been proposed to decouple DIFT from the program runtime [99, 37, 36, 12, 11]. For example, Taintpipe [12], Straight-taint [36] and ShadowReplica [11] pre-compute propagation models from the program source and use them to speed up the DIFT at runtime. However, their dependency on program source disables these systems to analyze undefined behavior. In contrast to these DIFT systems, RTAG provides both efficient runtime (recording) and the ability to reliably replay and perform DIFT on the undefined behavior (e.g., memory corruptions) commonly seen in recent attacks. Jetstream [37] records the normal runtime execution and defers tainting until replay by

Table 4.5: Comparison of DIFT-based provenance systems.

DIFT Systems	Cross Host	Inst Time	Tag Dep	Run Over	DIFT Over(T/M)
Dytan [32]	×	Runtime	Inlined	High	High/High
DataTracker [45]	×	Runtime	Inlined	High	High/High
Panorama [34]	×	Runtime	Inlined	High	High/High
ShadowReplica [11]	×	Runtime	Inlined	High	Low/High
Taintpipe [12]	×	Runtime	Inlined	High	Low/High
Panda [43, 69]	×	Replay	Inlined	High	High/High
Arnold [9]	×	Replay	Inlined	Low	High/High
RAIN	×	Replay	Inlined	Low	High/High
Jetstream [37]	×	Replay	Inlined	Low	Low/High
TaintExchange [70]	✓	Runtime	Inlined	High	High/High
Cloudfence [71]	✓	Runtime	Inlined	High	High/High
RTAG	✓	**Replay**	**Decoupled**	**Low**	**Low/Low**

splitting an application into several epochs. DTAM [100] uses dynamic taint analysis to find the relevant program inputs to its control flow and has a potential to reduce the workload of a record-replay system. Similar to RTAG, TaintExchange [70] and Cloudfence [71] provide multi-host information-flow analysis at runtime, but incur significant overhead (20× in some cases). We summarize the comparisons between RTAG and previous DIFT-based provenance systems in Table 4.5. "**Cross Host**" tells whether the system covers cross-host analysis; "**Inst Time**" represents when the instrumentation is performed (i.e., runtime or replay); "**Tag Dep**" shows how the tag dependency is handled; "**Run Over**" shows the runtime overhead; "**DIFT Over(T/M)**" presents the overhead of performing DIFT in terms of **T**ime and **M**emory cost in which RTAG both achieves reductions significantly.

Provenance Capturing. Using data provenance [101] to investigate advanced attacks, such at APTs, has become a popular area of research [72, 18, 102, 19, 23, 16, 103, 104, 21]. For example, the Linux Audit System [18], Hi-Fi [102], and PASS [19] capture system-level provenance with less than 10% overhead. Linux provenance modules (LPM) [22] allows developers to develop customized provenance rules to create Linux Security Modules and LSM-like modules. SPADE [21] decouples the generation and collection of provenance data to provide a distributed provenance platform, and ProvThings [105] generates provenance data for IoT devices. Unfortunately, these systems are restricted to coarse-grained prove-

nance, which generate many false dependencies. To reduce false positives and logging sizes, Protracer [16] improves BEEP [23] to switch between unit-level tainting and provenance propagation. In contrast, MCI [104] determines fine-grained dependencies ahead-of-time by inferring implicit dependencies using LDX [103] and creating causal models. DataTracker [45] leverages DIFT to provide fine-grained data, but incurs significant overhead. Finally, RAIN uses record and replay to defer DIFT until replay, then uses reachability analysis to refine the dependency graph before tainting. However, none of these systems can provide fine-grained cross-host provenance like RTAG because they have no tag association mechanism to support cross-host DIFT.

Network Provenance. In addition to system-wide tracking, provenance at network level is a well-researched area [78, 49, 79]. For example, ExSPAN [78] provides a distributed data model for storing network provenance. One challenge network provenance faces is that it obviously cannot detect most system-level causality on end nodes. Technically, network provenance and RTAG are orthogonal to each other, so that we can use both approaches together to further enhance attack detection.

Record Replay System. Deterministic record-and-replay has been a well-researched area [106, 107, 41, 85, 108]. In addition to providing faithful replay, the current state-of-the-art techniques allow instrumentation of programs during the replay of execution [9, 43, 42]. Arnold [9] provides efficient runtime because it is a kernel based solution and can efficiently record nondeterministic events. Aftersight [42] and PANDA [43] are hypervisor-based solutions. Aftersite is based on VMware hypervisor (record) and QEMU (replay) while PANDA is purely based on QEMU. Similar to RAIN, RTAG leverages Arnold to provide efficient recording performance, however the goals and functionality of RTAG are unique from to Arnold and could be implemented on other systems.

4.10 Conclusion

When investigating information flow-based cross-host attacks, analysts need to manually analyze the information flow generated by the processes running on multiple hosts. This is a time consuming, error prone, and challenging task, due to the high number of processes and consequently flows involved. To help analysts in this task, we propose RTAG, a system for accurate and efficient information flow analysis that makes cross-host attack investigation practical. We implemented and empirically evaluated RTAG by using the system to analyze a set of real-world attacks including GitPwnd, a state-of-the-art cross-host data infiltration attack. The system was able to provide accurate results while reducing memory consumption by 90% and also reducing the time consumption by 60-90% compared to related work. We have a plan to release the source code of RTAG.

We foresee several directions for future work. First, we plan to make hosts running RTAG interoperable with hosts not running the system. To do so, we plan to embed tag information in an optional field of the UDP header. Second, we plan to identify information flow techniques that are resilient to the fact that RTAG might not be running on every host in a given network. Third, we plan to integrate in-process parallel DIFT techniques to RTAG to further optimize the analysis time. Fourth, we plan to reduce the storage requirement for non-deterministic inputs. To do so, we plan to investigate ways to optimize the storage of similar executions across different hosts. Finally, we plan to extend the queries supported by RTAG so that it is possible to compare the information flow associated with different executions of the same program. In this way, it will be possible to pinpoint when and where a program was compromised.

CHAPTER 5

ADVERSARIAL ENGAGEMENTS BY DARPA

To show that our approach creates a sound, reliable, and real capability to handle these attacks, we opened our system to assault by a professional red-team group. Our systems managed to survive the attacks without data loss and provide data that reconstructed the attack behavior with high accuracy. Beyond testing RAIN and RTAG in sandboxed environments, we further test them in the real-world adversarial campaign funded by DARPA where a professional red team perform various attacks against our systems. Using the data generated by our system, the analyst is able to detect the attacks and reconstruct the provenance of them with high accuracy.

The Transparent Computing program. The Transparent Computing (TC) program aims to test the state-of-art system monitoring tools and analysis methods for investigating stealthy cyber attacks. In the program, a set of "system" teams provide customized systems with monitoring capabilities that generate data for analysis; meanwhile another set of "analysis" teams consume and analyze the data for attack investigation and forensic analysis. We are selected to participate the program working as one system team. The program has arranged five adversarial engagements for evaluation. In each engagement, the red team exercise offensive attacks against a target system where the monitoring tool is deployed. They try to stay low-key leaving no trace: they loaded themselves reflectively into memory to avoid executing from disk; system calls were leveraged to avoid commonly monitored APIs such as file `open()` and `write()`; modules were loaded in-memory for network recon, screen capture, audio capture, video capture, and keylogging; and privilege escalation and process injection are introduced to pivot to other processes.

Each engagement takes around two weeks, during which benign activities are performed on the target host mimicking a typical workload of desktop usage. The red team then

randomly performs attacks on the target hosts which blends with the benign data. The analysis team consumes the stream of system events generated on the target hosts and conducts real-time detection and analysis. The red team evaluate the system and analysis teams in terms of investigation accuracy after each engagement.

Data Model and System Setup. Since the analysis teams work with different system teams for consuming the data, the program defines a common data model (CDM) to unify the data structure and interface. Every team follows this model so the data consumption and sharing becomes easier. The CDM defines subjects, objects, and various system events (e.g., `read()`, `execve()`), and a UUID to identify subject/object and event. We internally also refer to this UUID system to identify the nodes and edges in our provenance graph. By doing this, the analysis teams can query to us using the UUID. We set up the target hosts as KVM instances, and an analysis host populating the provenance graph and translating the data into CDM. The data are shared to the analysis team using the kafka and zookeeper infrastructure in real time. We also have a RESTful API to support the query interface for the analysis team to query for the fine-grained analysis results. When a query arrives, we run reachability analysis and dispatches the replay and taint work to one of the replayers. The result is returned back to the analysis team using CDM.

Results. We survived the five adversarial engagements by DARPA, and achieved high detection rate and analysis accuracy. The results in TC supplement the evaluation contribution of this thesis. We present the results of analysis using the data generated by by our system. In Table 5.1, we show the number of attacks performed by the red team, the number of attacks whose behavior is captured by our system, and the number of attacks that are detected by the analysis team using our data. For counting the "capture" number, we compare our data and the ground truth, and look for events related to each attack. For the "detect" number, we count the total number of detection from both analysis teams consuming our data. Each analysis team does not detect all the attacks, but they collectively detect all the attacks, which indicates the effectiveness of our system as well as the high accuracy.

Table 5.1: Results of DARPA adversarial engagements

Adversarial Activity	# Performed	# Captured	# Detected
SSH Dropbear	3	3	6
Nmap Scanning and SCP	2	2	2
APT Simulation	3	3	6
Pine Metasploit	1	1	2
Phishing Email	2	2	4
Firefox Drakon APT	2	2	2
Micro APT	2	2	3

The description of each adversarial activity is in the Appendix A.

Lessons learned. We learned a lot from the engagement experience about how the detection works, how our data at either coarse-level or fine-grained level contributes to the detection accuracy, and the limitation of our system in the real-world environment. First, we find that monitoring at system-call level can effectively detect most of the attacks. Even if the attack relies on dynamic loading of payload, the analysis team can still catch the behavior via system calls such as `mmap` followed by a `mprotect` with setting the protection to `EXECUTIBLE`. Second, the analysis accuracy starts to decrease when dealing with attacks that have a pattern similar with benign activity. Since the analysis team trained their model based on a benign data generation period, their accuracy decreases when the attackers used benign program for malicious purpose (e.g., using `SCP` to download a malicious program). Third, using our fine-grained query interface helps remove false positive in data flow analysis. For example, our DIFT-based fine-grained analysis ruled out a data leakage case when a malicious program read from `/etc/hosts` and sent some data to a remote host. This would be a false positive detection if the analysis only looked at the system call sequence. Fourth, due to the time cost in DIFT, our fine-grained analysis cannot support real-time detection. The analysis team chose to use the system-call level data for the real-time detection section of the engagement instead. This indicates that our fine-grained analysis better fits the purpose of postmortem forensic analysis.

CHAPTER 6
EXPLORATION ON NEW RECORD REPLAY SYSTEM

6.1 Introduction

The record replay technique plays the key role in the refinable attack investigation system. Systems like Arnold [9] are able to reconstruct the full program states at single instruction granularity, which is essential for investigating memory related exploits. Though the record replay technique achieves low-overhead recording, the fine-grained forensic analysis for auditing the replayed execution is unnecessarily slow, because the replay still needs to go through the instrumentation with high overhead at the "delayed" time. We aim for a more efficient fine-grained forensic auditing of replay executions.

We envision a customizable framework that can be tuned for different analysis purposes (e.g., data flow tracking, data race detection). For the example of a record replay for data flow analysis, the framework only needs to record the data flow so we can guarantee to recreate it during replay time. The data flow analysis is sufficient for most popular forensic analysis. Besides the reduction of recording and replaying overhead, the new approach has the potential in improving the data privacy because we can selectively skip recording some data containing sensitive information.

6.2 Design

For the new record replay system, we transform a program into a "record" version and a "replay" version. The record version includes instrumentations needed to replicate the control flow and the data for analysis in replay. The replay version enforces the recorded control flow and offers the recorded data at specific locations (e.g., at certain instruction). The system works as a framework supporting extensible analysis. For example, we can

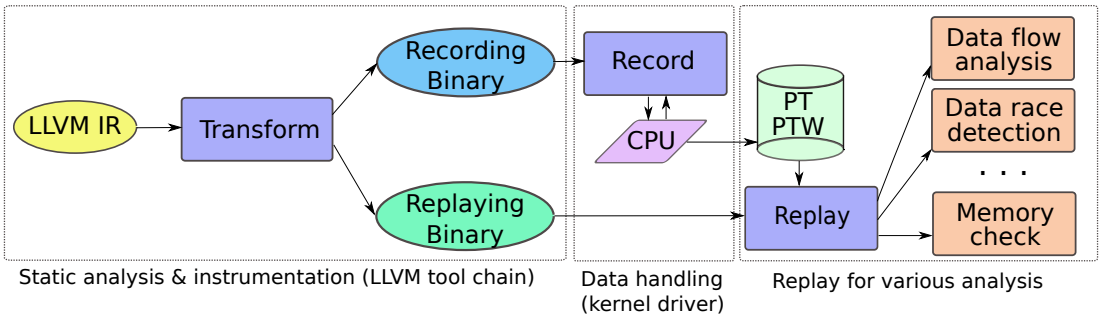


Figure 6.1: Overview of new system

port the DFSan to the framework by replaying the execution and providing the data to DFSan. Similarly, data race detection such as ThreadSan [109] can be ported by letting the framework only replay the data access interleavings and synchronizations. We depict the architecture overview of the new system in Figure 6.1.

Recorder. We record certain data during recording according to the analysis we want to perform at replay time. By replaying these recorded data directly, the program does not need to replay every program state in order to recreate the desired program states for the analysis. For example, for a data flow analysis, it is sufficient to replay the data flow if the loading of pointer is recorded.

Replayer. We transform the program into a replayer by constructing the "skeleton" of the program — only the recorded control flow is enforced at the replay. At the base level, the replayer works as a control flow replayer. Then, we selectively add the instructions into the replayer for the needs of analysis. For example, for DFSan as analysis, we record the address of the pointer being loaded, then replay the address at replay time. The recorded address will be presented at the corresponding IR instruction for the DFSan to track data flow. Other executions such as the external library calls are skipped since they are not related to the analysis (e.g., data flow).

6.3 Implementation

Kernel module for PT/PTW. We explore using a new hardware feature called Intel Processor trace and Processor trace write for the recording. Intel released Processor Trace (PT) as a built-in feature for obtaining the control flow of the processor for debugging purposes. A variety of work have demonstrated improvement of traditional analysis such as CFI thanks to the runtime efficiency of PT (e.g., [110, 111, 112]). Intel Processor Trace Write (PTW) was released recently to enhance the debugging capability on top of PT. Intel adds PTW as a new intrinsic into the x86 and x86_64 instruction set. For the 64bit, PTW can accommodate an 8-byte long data payload. The developer needs to instrument the program using PTW instructions. Compared with traditional compiler-level instrumentation tools, the PTW instruction provides essential debugging data more efficiently. To make PT and PTW available, Linux kernel adds the data transferring and parsing utility into the `perf` functionality. Unfortunately, there exists data loss issues in `perf`, so we develop a kernel module for data handling facility.

We implemented the kernel module to transfer the PT and PTW data from CPU to the file system. The CPU has an internal buffer for storing the PT and PTW data. The size of the buffer is limited so one needs to copy the data out to memory. If the data is not copied out in time, data overflow could happen such that new data pollute the old data that is still in the buffer. To ensure the copying is in time and frequent enough, we rely on the interrupt handling routine to trigger this copying. The copyed data are stored in the kernel memory (i.e., `kmem_cache`). We use a work_queue to flush the data from the kernel memory to persistence.

Program transformation. We implemented an LLVM IR level tool to generate the recorder and replayer. The recorder adds instrumentation using PTWrite intrinsics the loading of pointers. The replayer enforces the control flow by reading the PT data and determining the branch result. Replays of data (e.g., loading of pointer) is done by reading

the PTW data.

Closing the gap between IR and binary. Since our tranformation works at LLVM IR level, while the PT/PTW data comes from the the X86 binary level, we need to fix the mismatch between these two levels. For example, the destinations in the conditional branch at IR level can lead to different cases at binary level. The code gen can yield the changed order of jump-to and fall-through destinations. We automated the comparison between the IR and the binary and handle the difference in the transformation.

6.4 Lessons learned

As soon as we have the kernel module and the program tranformation implemented, we find a technical problem in recording. Basically, when testing the SPEC CPU benchmark or the Splash multi-thread benchmark, we find two types of data loss. First, there is data overflow inside the CPU buffer. This is due to intensive PTWrite operations such that if the distance between two PTWrite is too short, the internal buffer inclines to get overflowed. Second, there is out-of-memory issue for the kernel memory. Basically, the data of PT and PTW are dominating the use of kernel memory such that the kernel becomes out of memory. This results in a `kmem_cache_alloc` error. These two issues are both due to high volume of data generated by PT/PTW, mainly PTW. In order to resolve these, we tried different approaches.

Slowing down. For intensive PTW operations, we strive to slow down the program. We add spins between two close PTW operations, so the kernel has more chance to ship the data out before leaving the buffer full. We do a spin adding based on the profiling result showing the distance between each two PTWrite, and add spins to the hottest and most intensive PTWrite. This alleviates the data loss cases, but does increase the recording overhead.

Optimization. We also attempted to reduce the data from the origin. We optimized the recording by moving the PTWrite out of a loop if it is invariant. We also search for and record the origin of pointers instead of uses. We do a backward traversal from the use of pointer and search for the path that has least overhead in recording. Then at the replay

time, we replay the origin, and the instructions leading to the use of pointer, the pointer is recovered at the same location of use as previously, but the recording overhead is reduced. This optimization helped reduce the recording overhead, but the data loss still exist in many benchmark items.

6.5 Hybrid Approach

Currently, we are exploring a hybrid approach that both selectively records program inputs (i.e., for replay states recreation) and directly records certain program states for analysis. For recording program inputs, we record most of external function returns and side effects. For recording program states directly, we run static analysis and dynamic profiling to determine which record states will be needed in order for the desired replay states to be present in the analysis. Based on the record states, we instrument the program accordingly. By doing this, we can accelerate the replayed execution without the need to recreate every program state.

CHAPTER 7
CONCLUSION

In this thesis, I have defended the thesis statement by demonstrating how refinable attack investigation achieves both high efficient runtime of the program and the fine-grained program behavior at the instruction level, even in today's complex multi-host systems. I first presented a record-replay-based attack investigation system that can achieve instruction-level data as well as an efficient runtime. Then I presented a tagging mechanism that extends the scope of the framework to multiple hosts that have data dependency with each other. Refinable investigation enables a strong postmortem analysis such that the analyst can time travel to a past moment to replay an execution with fine-grained analysis.

The current implementation of the systems resides inside the Linux kernel. Typical kernel protection methods need to be implaced to avoid the kernel from being compromised. Also, the deployability is limited as one needs to port the system to another version of kernel if necessary. We envision a userspace based implementation is possible such as the RR [59] project. Another direction would be to add application specific instrumentations to complex programs. Since system-call level events are too low level for such applications, in order to reason the data causality and attack procedure, application specific data at higher level are needed. For example, instrumenting the access of DOM tree elements and javascript-level objects can help understand more browser specific attacks such as phishing, cross-site scripting (e.g., [113, 114]). Combining the refinable capability with these higher level program semantics is potentially interesting.

Appendices

APPENDIX A

ADVERSARIAL ACTIVITIES

SSH Dropbear. Dropbear is a widely used SSH server available in FreeBSD and Ubuntu package managers [115]. The modification is a hardcoded key, with a comparison for "ta5rules" in `svr_auth_password()` of `svr-authpasswd.c`. The second inserts a hardcoded public key into dropbear's stream when it reads the authorized keys file in `svr-authpubkeys.c`. This has the effect that any attacker with the private key gains access without a password, and the authorized keys file is not modified.

Nmap Scanning. The red team implements a couple rudimentary network scan features reinforcing the network recon and pivoting capability. The tool avoids simply including an nmap binary since it is often identified as an attacker's tool.

APT Simulation. APT Simulation uses TCP communication for C2 and supported getfile, putfile, execfile, and shell commands. Commands are sent via messages created using Google Protocol Buffers with the nanopb plugin to minimize size. Modules are deployed to the target by writing them to disk and executing them. A new shell process and connection out to the C2 server provides a remote shell capability, but all commands run in the shell (such as whoami, hostname) result in new processes created using commands from the target environment.

Pine Metasploit. Pine is patched to add a backdoor to the mail view code that checks for a specific file attachment named "tcexec". If the e-mail has an attachment that is an application with name "tcexec", the backdoor writes the attachment to file at `/tmp/tcexec`, changes the permissions to 777, forks a new process, and calls execve on "/tmp/tcexec". The attachment is expected to be an executable binary, such as drakon APT or micro APT, which is run as a new process and not in the memory of the Pine process. Additionally, the Pine patch also collects IMAP usernames and passwords and writes them to file `/tmp/tcexfil`.

Phishing Email. Spear phishing is a popular technique where the attacker sends an e-mail that looks legitimate to trick the user into doing something they should not to do to help the attacker achieve an objective. The goal of phishing is to get the user to unwittingly download and run the malware or to give away their credentials or personal information.

Firefox Drakon APT. Drakon APT improves upon APT simulation by introducing more advanced capabilities and avoiding actions known to be detected. Over the course of Transparent Computing program, Drakon APT moves increasingly more functionality into in-memory to avoid writing to disk and creating new processes. Drakon APT supports getfile, putfile, execfile, and shell commands but does not rely on them nearly as much, especially execfile and shell. In their place, Drakon APT uses built-in commands via syscalls and module loading bypassing the OS library loaders. Syscalls allows for the implementation of built-in commands such as `whoami, hostname, ls, ps, cd, pwd`, and others. By using syscalls directly, Drakon APT is able to avoid using well-known API functions such as `open`. If those API functions are being monitored, they will miss Drakon APT's use of that functionality. Other changes are made too in order to obfuscate attacks, including switching from TCP to HTTP with encrypted payloads. APT simulation uses unencrypted TCP which could be captured and analyzed by performers. Drakon APT sends HTTP requests and responses with image payloads where the image is actually an encrypted command request or response.

Micro APT. Micro APT is an attempt at a minimal size but still full featured APT simulacrum using its own C2 interface instead of Drakon APT's OC2. It is built with mostly the same functionality as Drakon APT but without Google Protocol Buffers, pthread, etc in order to significantly reduce size. Additionally, Micro APT adds support for running Python scripts on the target, scanning the network using nmap. Because of Micro APT's minimal size, it is an ideal tool to use for testing new capabilities as integration requires minimal time and effort.

REFERENCES

[1] *Equifax data breach*, https://en.wikipedia.org/wiki/Equifax, Oct. 2017.

[2] *Marriott data breach*, https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach, Dec. 2018.

[3] *Sony data breach*, https://www.washingtonpost.com/news/the-switch/wp/2014/12/18/the-sony-pictures-hack-explained/, Dec. 2014.

[4] *Yahoo data breach*, https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html, Oct. 2017.

[5] *Gemalto breach index*, https://breachlevelindex.com, Oct. 2018.

[6] *Google and facebook accused of manipulating users into giving up their data*, https://fortune.com/2018/06/27/google-facebook-dark-patterns-privacy, 2018.

[7] *Advanced persistent threat*, https://www.cybereason.com/blog/advanced-persistent-threat-apt, Jan. 2018.

[8] *Fileless malware*, https://www.cybereason.com/blog/fileless-malware, Dec. 2017.

[9] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[10] *Sysdig*, https://www.sysdig.org, Oct. 2017.

[11] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[12] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[13] C. Gibler and N. Beddome, *Gitpwnd*, https://github.com/nccgroup/gitpwnd, Oct. 2017.

[14] *Intel processor trace*, https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, Oct. 2017.

[15] *Dtrace*, https://www.dtrace.org, Oct. 2017.

[16] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[17] *Snort*, https://www.snort.org, Oct. 2017.

[18] *Linux audit*, https://linux.die.net/man/8/auditd, Oct. 2017.

[19] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2006.

[20] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jun. 2009.

[21] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.

[22] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[23] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[24] *Ssh 1.2.x - crc-32 compensation attack detector*, https://www.exploit-db.com/exploits/20617, Feb. 2001.

[25] *Openssh 3.x - challenge-response buffer overflow*, https://www.exploit-db.com/exploits/21578, 2002.

[26] *Use after free*, https://www.owasp.org/index.php/Using_freed_memory, Oct. 2018.

[27] *Integer overflow*, https://cwe.mitre.org/data/definitions/190.html, Oct. 2018.

[28] *Intel pin*, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, Oct. 2017.

[29] *Dynamorio*, https://www.dynamorio.org, Oct. 2017.

[30] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2005.

[31] G. E. Suh, J. W. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Oct. 2004.

[32] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, Jul. 2007.

[33] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, London, UK, 2012.

[34] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2007.

[35] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[36] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: Decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, Sep. 2016.

[37] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "JetStream: Cluster-scale parallelization of information flow queries," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[38] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[39] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2004.

[40] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[41] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.

[42] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2008.

[43] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)*, 2015.

[44] M. Stamatogiannakis, P. Groth, and H. Bos, "Decoupling provenace capture and analysis from execution," in *Proceedings of the 7th USENIX Workshop on the Theory and Practice on Provenance (TaPP)*, Edinburgh, Scotland, 2015.

[45] ——, "Looking inside the black-box: Capturing data provenance using dynamic instrumentation," in *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*, Cologne, Germany, 2014.

[46] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical data protection for emerging iot application frameworks," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[47] *DARPA transparent computing program*, `http://www.darpa.mil/program/transparent-computing`, Oct. 2017.

[48] K. Baumgartner, *On the StrongPity waterhole attacks targeting Italian and Belgian encryption users*, `https://securelist.com/blog/research/76147`, Oct. 2017.

[49] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[50] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let sdn be your eyes: Secure forensics in data center networks," in *2014 NDSS Workshop on Security of Emerging Network Technologies (SENT)*, 2014.

[51] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2004.

[52] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integirty for commodity OSes," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[53] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[54] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[55] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "KI-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[56] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: Detecting cache-based covert channel attacks using record and replay," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[57] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, "Detecting covert timing channels with time-deterministic replay," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[58] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[59] *Mozilla rr*, `http://rr-project.org`, Oct. 2017.

[60] *Apache avro*, `https://avro.apache.org`, Oct. 2017.

[61] *Neo4j graph database*, `http://neo4j.com`, Oct. 2017.

[62] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[63] *Squid*, `http://www.squid-cache.org`, Oct. 2017.

[64] E. A. Manzoor, S. Momeni, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, 2016.

[65] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A framework for user-intent monitoring of text-based networked applications," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[66] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[67] T. Kim, R. Chandra, and N. Zeldovich, "Recovering from intrusions in distributed systems with DARE," in *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, Jul. 2012.

[68] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS'16)*, 2016.

[69] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[70] A. Zavou, G. Portokalidis, and A. Keromytis, "Taint-exchange: A generic system for cross-process and cross-host taint tracking," in *Advances in Information and Computer Security*, Springer, 2011, pp. 113–128.

[71] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis, "Cloudfence: Data flow tracking as a cloud service," in *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Saint Lucia, Oct. 2013.

[72] Y. Ji, S. Lee, and W. Lee, "RecProv: Towards provenance-aware user space record and replay," in *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*, Mclean, VA, 2016.

[73] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou, "Data provenance at internet scale: Architecture, experiences, and the road ahead," in *Conference on Innovative Data Systems Research (CIDR'17)*, Chaminade, CA, 2017.

[74] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *Proceedings of the 2016 ACM SIGCOMM*, Florianopolis, Brazil, Aug. 2016.

[75] P. Chen, L. Desmet, and C. Huygens, "A study on advanced persistent threats," in *IFIP International Conference on Communications and Multimedia Security*, Springer, 2014, pp. 63–72.

[76] *Git: A free and open source distributed version control system*, `https://git-scm.com/`, Feb. 2018.

[77] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "RAIN: Refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.

[78] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," in *Proceedings of the 2010 ACM SIGMOD/PODS Conference*, ACM, Indianapolis, IN, Jun. 2010, pp. 615–626.

[79] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," in *ACM SIGCOMM Computer Communication Review*, vol. 44, Snowbird, Utah, USA, Jun. 2014, pp. 383–394.

[80] *Event tracing for windows*, `https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/etw-tracing`, Oct. 2017.

[81] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," 2005.

[82] Y. Saito, "Jockey: A user-space library for record-replay debugging," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005.

[83] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[84] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*, 2013.

[85] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr, "Abstractions for practical virtual machine replay," in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Atlanta, GA, 2016.

[86] *Gitolite*, `https://www.gitolite.com`, Feb. 2018.

[87] *Gtk-gnutella*, `http://gtk-gnutella.sourceforge.net`, Oct. 2017.

[88] *Taintgrind: A valgrind taint analysis tool*, `https://github.com/wmkhoo/taintgrind`, Feb. 2018.

[89] J. Postel, "Transmission control protocol," 1981.

[90] ——, "User datagram protocol," 1980.

[91] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *ACM SIGPLAN Notices*, ACM, vol. 49, 2014, pp. 227–238.

[92] *Postgresql*, `https://www.postgresql.org`, Oct. 2014.

[93] *X86 machine code statistics*, `https://www.strchr.com/x86_machine_code_statistics`, Oct. 2014.

[94] *Kernel-based virtual machine*, `https://www.linux-kvm.org`, Oct. 2018.

[95] *Linux network file system*, `http://nfs.sourceforge.net/`, Oct. 2018.

[96] *Peer-to-peer peril: How peer-to-peer sharing impacts vendor risk and security benchmarking*, `https://info.bitsighttech.com/how-peer-to-peer-`

file - sharing - impacts - vendor - risk - security - benchmarking, Dec. 2015.

[97] *Iperf3*, https://iperf.fr, Feb. 2018.

[98] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, ACM, Vienna, Austria, Oct. 2016, pp. 331–342.

[99] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan, "Parallelizing dynamic information flow tracking," Munich, Germany, 2008.

[100] M. Ganai, D. Lee, and A. Gupta, "Dtam: Dynamic taint analysis of multi-threaded programs for relevancy," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Cary, NC, Nov. 2012.

[101] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, Jun. 2005.

[102] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: Collecting high-fidelity whole-system provenance," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 259–268.

[103] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "LDX: Causality inference by lightweight dual execution," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[104] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[105] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[106] D. F. Bacon and S. C. Goldstein, *Hardware-assisted replay of multiprocessor programs*. Santa Cruz, CA, 1991.

[107] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "Dmp: Deterministic shared memory multiprocessing," in *ACM SIGARCH Computer Architecture News*, ACM, New York, NY, 2009.

[108] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '10, New York, NY, 2010.

[109] *Threadsan*, `https://clang.llvm.org/docs/ThreadSanitizer.html`, Oct. 2017.

[110] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse debugging of failures in deployed software," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Nov. 2018.

[111] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[112] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[113] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[114] B. Q. Li, P. Vadrevu, K. H. Lee, and R. Perdisci, "Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[115] *Dropbear*, `https://community.rsa.com/thread/186012`, Oct. 2017.

Curriculum Vitae

Yang Ji

Room 3110, 266 Ferst Dr., Atlanta, GA, 30326
(404) 906 7018, yang.ji@gatech.edu

EXPERTISE Operating systems and software security

EDUCATION

Georgia Institute of Technology, Atlanta, United States *Aug. 2014 - Present*
Ph.D. in computer science
- Research area: systems and software security
- Advisor: Dr. Wenke Lee

Seoul National University, Seoul, Republic of Korea *Sept. 2007 - Aug. 2009*
M. Sc. in electrical engineering and computer science
- Thesis: reduced communication complexity of batch key distribution (best thesis award runner-up)
- Advisor: Dr. Seung-Woo Seo

Shanghai Jiao Tong University, Shanghai, China *Sept. 2003 - June 2007*
B. Sc. in information security
- Thesis: implementation of m-ary covert channel using Linux socket programming

RESEARCH PROJECTS

Multi-level refinable information flow tracking (Lead)
GTISC, Georgia Institute of Technology *Mar. 2015 - Present*

- Prototyped a refinable information flow tracking system that relies on Linux kernel instrumentation-based record-replay technique and dynamic-instrumentation-based analysis.
- Evaluated the system in the DARPA Transparent Computing program and achieved: efficient runtime performance (less than 5% overhead) and instruction-level fine-grained provenance that existing provenance systems cannot provide.

Distributed in-browser social network (Lead)
Systems Software & Security Lab, Georgia Institute of Technology *Aug. 2014 - Feb., 2015*

- Prototyped a distributed in-browser social network using WebRTC. The system used a trust net of friendship based on signed public keys, piggybacked user credential management on public key infrastructures, and applied Distributed Hash Table (DHT) as overlay network for data synchronization.

Efficient batch key distribution (Lead)
Computer Networks and Security Lab, Seoul National University *Sept. 2008 - Aug. 2009*

- Derived and proved the communication cost lower bound of batch key distribution in the multicast security area. Proposed two novel protocols that ran close to the optimal bound and achieved notable scalability enhancements in terms of communication and storage cost.

PUBLICATIONS

Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.
Ruian Duan, Ashish Bijlani, **Yang Ji**, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee.
In *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking.
Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alex Orso, and Wenke Lee.
In *Proc. Usenix Security Symposium (Security)*, Baltimore, MD, 2018.

Improving Accuracy of Android Malware Detection with Lightweight Contextual Awareness.
Joey Allen, Matthew Landen, Sanya Chaba, **Yang Ji**, Simon Chung, and Wenke Lee.
In *Proc. Annual Computer Security Applications Conference (ACSAC)*, Puerto Rico, USA, 2018.

RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking.
Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alex Orso, and Wenke Lee.
In *Proc. ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques.
Meng Xu, Chengyu Song, **Yang ji**, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim.
In *ACM Computing Surveys (CSUR)*, 2016.

RecProv: Towards Provenance-Aware User space Record and Replay.
Yang Ji, Sangho Lee, and Wenke Lee.
In *Proc. International Provenance and Annotation Workshop (IPAW)*, Washington DC, 2016.

Optimizing the Batch Mode of Group Rekeying: Lower Bound and New Protocols.
Yang Ji and Seung-Woo Seo.
In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, San Diego, CA, 2010.

DAKS: An Efficient Batch Rekeying Scheme for Departure-Aware Multicast Services.
Yang Ji and Seung-Woo Seo.
In *Proc. IEEE Global Telecommunications Conference (GLOBECOM)*, Honolulu, HI, 2009.

WORK EXPERIENCE

NEC Labs America, Princeton, New Jersey *May 2015 - Aug. 2015*

Summer Intern, Autonomic Security Intelligence (supervised by Dr. Fengyuan Xu)

- Built a latest state query service at the centralized monitoring server that aggregates the system level events from a large scale of hosts. Patternized the normal behavior of users in order for anomaly detection.

Samsung Electronics, Suwon, Republic of Korea *Sept. 2009 - July 2014*

Software Engineer Lead, LTE Access Network S/W Lab, Telecom. Sys. Division
Operations and Support System (OSS) (*Sept. 2010 - July 2014*)

- Designed and implemented high-level OSS protocols that control radio units in the large-scale LTE base stations. The protocols are widely deployed by a number of telecom operators such as Verizon, Sprint, Clearwire, KDDI(Japan) and SKT(Korea).

Software Engineer, Advanced Communication Solution Lab, DMC R&D Center
Element Management System (EMS) (*Sept. 2009 - Aug. 2010*)

- Developed a centralized network element management system prototype that provides a graphic interface of device/cell configuration, fault management, self-diagnosis and real-time performance analysis. The system with its extensions is currently used by major telecom operators.

PATENTS

"System and method for batch rekeying", Korean Patent, No.1099851, with Seung-Woo Seo.

TALKS

"Key management schemes in LTE: Challenge and Outlook", 10^{th} Samsung Global Software Confer-

ence, Yongin, South Korea, Oct 2010. (Invited Talk)
"Distributed Social Network", NEC Lab North America, Princeton, NJ, May 2015.
"Refinable Attack Investigation", CCS 2017, Dallas, TX, Oct 2017.
"Refinable Attack Investigation", Georgia Tech Demo Day Final, Atlanta, GA, April 2018.
"Enabling Cross-Host Refinable Attack Investigation", Usenix Security 2018, Baltimore, MD, August 2018.
"Enabling Cross-Host Refinable Attack Investigation", DARPA Transparent Computing PI meeting, Washington DC, August 2018.

HONORS &AWARDS

Bronze Medal of Samsung Best Paper Contest (Top 1%)	*2010*
Samsung Global Scholarship (5 recipients out of 5000+ applicants)	*2007 - 2009*
Runner-up of Best Thesis Award in Seoul National University (Top 5%)	*2009*
Excellent Prize of Microsoft Software Component Design Contest, China (Top 2%)	*2006*
Creativity Prize of Shanghai Embedded System Design Contest, China (Top 3%)	*2006*
Excellent Academic Scholarship of Shanghai Jiao Tong University (Top 10%)	*2004 - 2006*

SKILLS

- Programming languages and tools: C, C++, LLVM, Intel Pin, Javascript, Python, Java
  Web development: Nodejs, Express, MongoDB, IndexedDB, HTML5, Selenium-Webdriver.
- Strong problem modeling and solving skills with expertise in using graph theory, queuing theory, convex optimization and operations research.
- Solid academic background in computer and network security: applied cryptography, access control, intrusion detection system (IDS), key management and authentication protocols.
- In-depth experience in OSS architecture and functionalities (configuration, fault, diagnosis and loading management) and good knowledge of 3G WiMAX and 4G LTE access network standards.
- **English** (Fluent, TOEFL iBT 110 with 27 in Speaking), **Korean** (Intermediate)